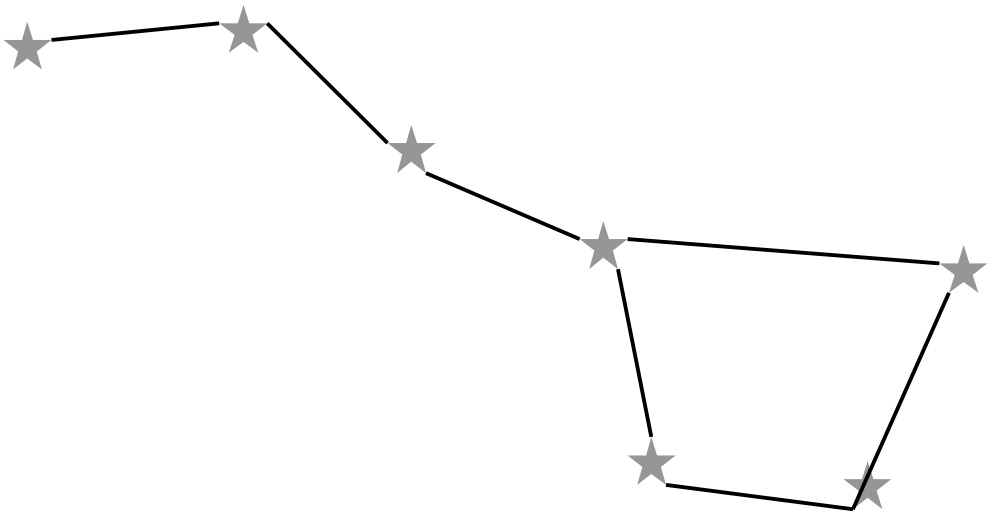
ГрафоМания



Редакция 1.0

от 2015-09-21

http://oleg-derevenets.narod.ru

Аннотация

Рассмотрены алгоритмы на графах и множествах. Неформальное изложение алгоритмов сопровождается работающим кодом с контрольными примерами, доведенными до числа. Код воплощен на объектно-ориентированном языке программирования **Delphi**. Подробно описана техника программирования, а листинги детально прокомментированы. Книга может служить дополнением к учебникам по дискретной математике и справочником по алгоритмам. Она будет полезна студентам, аспирантам и программистам, решающим практические задачи в этой области.

Оглавление

Глава 1 Когда я стану программистом?

«Когда я стану программистом? — втайне терзается новичок, — сколько языков мне освоить, и каких? Что востребовано теперь на рынке? Вот выучу *этот* язык, затем *тот*, а уж потом...». И что потом?

1.1. «Всё проходит»

Всё проходит — сокрушался мудрый Соломон. Будь он программистом, то поправил бы: *всё проходит очень быстро*. Ещё бы! Всего лишь 3 десятка лет тому пара языков программирования вкупе с небольшой практикой вполне законно давали вам титул программиста. Ныне же соискателей звания смущают, по меньшей мере, два вопроса, второй из которых таков: как долго я останусь программистом? Ведь новые языки всё плодятся и плодятся, а модные технологии и направления стремительно меняют ландшафт профессии. Только вчера ты освоил новинку и был «на коне», а сегодня ловишь на себе сочувствующие взгляды юных конкурентов. Коллеги не дадут соврать: удел программиста — грести против течения: чуть сбавил усилия, бросил вёсла, и уже влечёт тебя поток к опасным порогам. Где якорь, что остановит гибельное движение? Где опора о твёрдое дно бурлящего потока? Когда новичок станет программистом и как долго останется им?

Проявив каплю терпения, и одолев десяток страниц, вы сами ответите на этот важный для многих вопрос. Со стороны сфера программирования видится пышным раскидистым деревом. Да, некоторые его ветви заметно увяли, зато другие бурным ростом сулят заманчивые перспективы. И всё же это только ветви… Мы же докопаемся до корней. Не удивляйтесь, но начнём издалека.

1.2. История трёх революций

Прежде, чем двинуться к цели, выясним наше место положения. Где место программиста в современных технологиях? Мы найдём его через знакомство с тремя кибернетическими революциями.

1.2.1. Революция первая — биологическая

Она случилась миллионы лет тому с появлением первых примитивных микроорганизмов. Своё пропитание — энергию, они брали, главным образом, от солнца. А надо заметить, что уже тогда, задолго до рождения Дарвина, работали его законы: мутация и естественный отбор. Мутации — это случайные изменения организма под действием внешних факторов. Мутации могут быть и полезными, и вредными — вердикт о полезности выносит естественный отбор.

И вот, вследствие мутаций и отбора часть микробов приноровилась брать дополнительную энергию из останков своих соплеменников, что усилило их конкурентное преимущество, — так появились наши предки, и был сделан первый шаг в последующей истории. Известно, однако, что под лежачий камень вода не

течёт: мудрено прокормиться, сидя на месте. Второй шаг эволюции привёл к случайному появлению примитивных органов движения — ресничек. И естественный отбор утвердил новинку: ведь даже хаотические перемещения в среде обитания существенно обогащали рацион обладателей ресничек.

На третьем этапе эволюция сотворила примитивные органы чувств — своего рода датчики, которые воздействовали прямо на органы движения. Датчики могли реагировать на освещённость, температуру среды обитания или её химический состав. Движения организмов обрели некую степень разумности: примитивное животное двигалось не куда попало, а в направлении пищи или лучших условий обитания.

Позже развились промежуточные звенья, передающие сигналы от органов чувств к органам движения — прообраз нервной системы. Этот посреднический слой помаленьку занялся логической обработкой сигналов, что сделало животных «умнее». Нервная система усложнялась и совершенствовалась миллионы лет, что и породило современных животных и человека. В нервной системе (мозгу) животного отражены и среда существования, и его роль в этой среде. Если отражение окружающей действительности назвать *моделью*, то мозг и нервная система, безусловно, являются *биологической моделью* окружающей среды и роли в ней данного организма.

Отметим две особенности биологической модели. Во-первых, она передаётся только по наследству от родителей к потомкам: животные не могут передать генетический опыт иначе. Конечно, высшие животные иногда обучают потомство, организуют коллективную охоту и оборону, но это не меняет врождённую модель. Отсюда следует другая её особенность: биологическая модель крайне консервативна, она не успевает за резкими изменениями условий обитания.

1.2.2. Революция вторая: символические модели

Старт ей дали примитивные изображения животных и людей, вырубленные грубой рукой древнего художника. Восхищение от этих каракулей не уступало тому, что возбуждает заезжая рок-группа у обитателей провинции. Сегодня нам трудно их понять, но ведь тогда человек впервые нутром почуял возможность *самому* создать *модель* какой-то части окружающей его природы. Эта модель, в отличие от модели биологической, оказалась доступной всем зрячим соплеменникам художника: и живущим, и будущим поколениям. Модель можно было дополнять, копировать, развивать, и для этого не требовались миллионы лет эволюции или участие автора прототипа.

Первые письмена (египетские, например) были рядом картинок, восточные иероглифы, вероятно, также порождены рисунками. Позже возникли современные алфавиты, не связанные с изображениями чего-либо, а также цифры, математические знаки, нотная запись и т.д. Появились литература, науки, художества и музыка. Всё это — современная культура, которая, подобно древним рисункам, являет совокупность *моделей* окружающего мира, но, в сравнении с наскальным рисунком, куда более совершенных. Эти модели объединяет то, что

все они изображаются символами, и потому их называют *символическими*. Модели хранят в себе накопленные знания и служат, как и биологические модели, всё той же цели: обретению преимуществ в борьбе за выживание. Но теперь говорим о выживании не только отдельных людей или групп, речь о государствах и человечестве в целом.

Отметим, что культура развилась лавиноподобно: с момента её зарождения прошли не миллионы, а тысячи лет, особенно бурным выдалось предыдущее столетие. Однако накопившийся к середине 20-го века громадный объём знаний породил новую проблему.

1.2.3. Революция третья — «кибермозг»

Символические модели — науки, искусства — создаются людьми и оживляются ими же: даже самая удачная модель — научная теория — мертва без человека. Модель оживает только в головах тех, кто постиг её ценой обучения. Так, проявление моделей, связанных с созданием машин, возможно лишь через хорошо образованных людей: конструкторов, инженеров, рабочих.

Известно, что в деле инженера, наряду с творческой, немало и черновой работы: рутинных вычислений по известным формулам (а формулы — это тоже простые символические модели). Вычисления утомляют экономистов, бухгалтеров, и других работников умственного труда. К середине 20-го века с усложнением технологий эта рутина буквально задавила специалистов, не оставляя им времени на творчество. Разверглась пропасть между количеством и качеством научных знаний с одной стороны, и возможностью их применения с другой. Рождалось много в принципе полезных, но не осуществимых по причине их сложности моделей, — и научно-технический прогресс затормозился.

И тогда (не чудо ли?) на смену счётам явились электронные вычислительные машины (ЭВМ). Первые ЭВМ служили, главным образом, для громоздких вычислений по формулам. Не зря название одного из ранних языков программирования — ФОРТРАН — так и расшифровывается: ФОРмульный ТРАНслятор. Постепенно компьютеры и языки программирования стали приспосабливать к переводу в электронную форму всё более сложных символических моделей. И теперь компьютеры понимают речь, различают образы, процессоры спрятаны в телефонах, микроволновых печах и другой технике — всего не перечислить. Куда двинулся прогресс? — в сторону перевода *символических* моделей в *электронную* форму. И программист оказался в самой гуще этой третьей стремительной кибер-революции. Какова ж его роль?

1.3. Модели, модели, модели…

Прежде, чем уяснить роль программиста, бегло ознакомимся со свойствами и назначением того, что мы называем *моделями*.

1.3.1. Много ль в корыте корысти?

Любая модель (биологическая, символическая) нужна для *предсказаний*. Мозг хищника *предсказывает* поведение жертвы и строит на этом предсказании план охоты. А жертва пытается *предсказать* поведение хищника и планирует спасение. Мозг тренированного игрока *прогнозирует* полёт мяча. Такова роль биологических моделей.

Тому же служат и символические модели. Инженерные расчёты *предсказывают* поведение будущей машины или прочность постройки. Социально-экономические модели, пусть и не вполне точные, с некоторой вероятностью *предрекают* грядущее. Даже такая расплывчатая модель, как художественное произведение, даёт, наряду с эстетическим наслаждением, представление о сторонах жизни, с которыми мы не сталкиваемся повседневно. Входя в положение героев романа или фильма, мы невольно *прогнозируем* своё поведение в сходных обстоятельствах. Итак, назначение любой модели — *предсказание*.

1.3.2. Неформализованные модели: искусство и философия

Поскольку разного рода искусства обладают некоторой *предсказательной* силой, мы причисляем их к моделям. Художественные произведения обычно фиксируют на бумаге или на других носителях посредством символов: буквами, нотами; стало быть, эти модели — символические. Но воздействуют они на нас не символами, а *образами* или *звуками*, порождаемыми символами, — именно они вызывают эмоции.

К искусствам примыкает философия, она тоже выражается символами (словами) и оперирует с образами. Однако если объектом искусств являются *конкретные* образы, взятые из природы (люди, пейзажи, звуки и т.д.), то в голове философа рождаются *абстрактные* образы — идеи, отражающие, по его мнению, общие законы природы.

Искусства и философию объединяет их *неформализованность*. Известно, что машину не увлечёшь ни фильмом, ни музыкой, её «мозг» не породит философских мыслей. Да и зачем ей это? Оставим приятное себе.

1.3.3. Формализованные модели: описательные науки и математика

Обратимся теперь к точным наукам, к тому, что можно хотя бы отчасти *формализовать* и переложить на компьютерные плечи. Науки делят на описательные и теоретические, хотя чёткой грани тут нет. География, биология, химия — что это? Скорее описательные науки. Но чем больше в науке математики, тем сильнее она смещается в сторону теоретическую, — такова физика, например. Крайней гранью является сама математика, объекты которой в природе не встречаются, хотя и навеяны ею. Подобно идеям философов, они являют плод

размышлений математиков. Описательные науки соотносятся с теоретическими так же, как искусства соотносятся с философией: описательные исследуют конкретные объекты природы, а теоретические — абстрактные, воображаемые объекты (табл. 1-1).

Табл. 1-1 — Классификация наук в зависимости от языка и изучаемых объектов

Используемый язык	Характер объектов	
	Конкретные объекты	Абстрактные объекты
Неформализованный	*Искусство*	*Философия*
Формализованный	*Описательные науки*	*Теоретические науки*

1.3.4. Царица наук

Царица наук — это математика, конечно. Не будучи прямо привязана к реальным объектам природы, она, тем не менее, проникает во все естественные науки, включая описательные. Все теории, претендующие на сколь-нибудь точные предсказания, не ограничиваются словами, а выражаются строгим языком математики.

1.3.5. Универсальность и обратимость математических моделей

Рассмотрим следующую формулу:

A = B • C

Угадайте, какую природную закономерность она выражает? Подсказки ищите в таблице, где та же формула слегка перелицована и снабжена пояснениями:

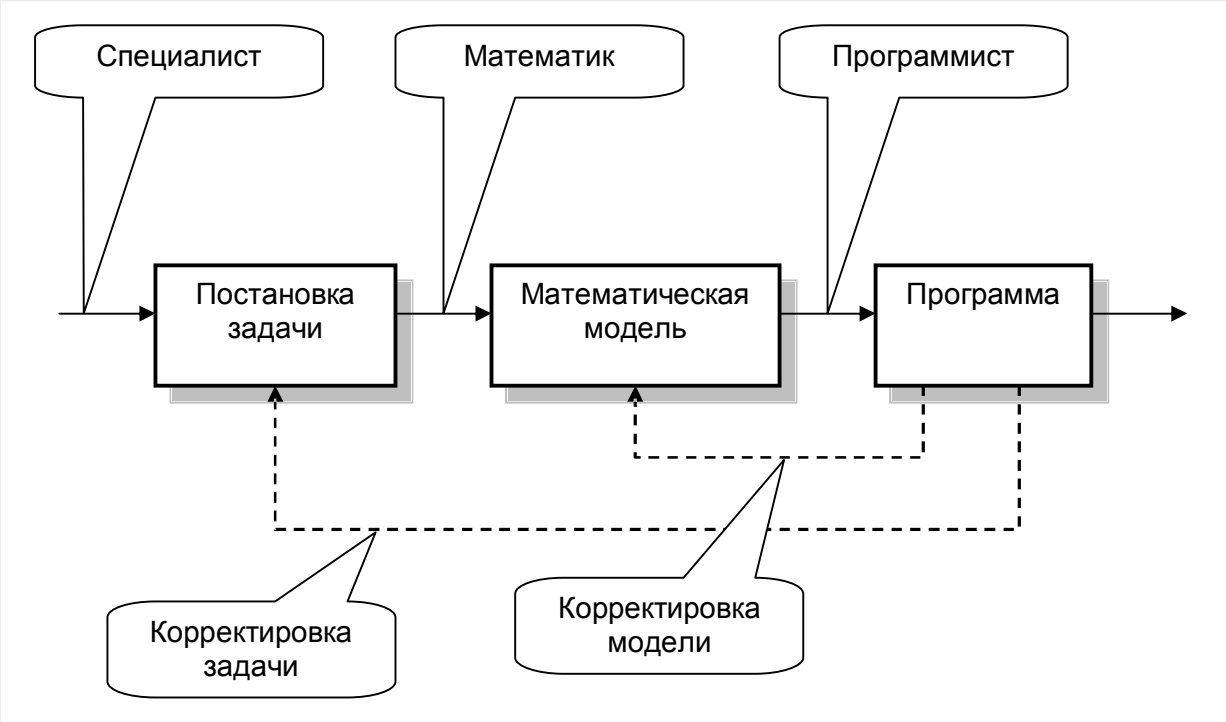
Формула	Обозначения
S = H • L	S — площадь прямоугольника H — его ширина L — его длина
L = V • T	L — преодолённый путь V — скорость T — время
S = C • N	S — стоимость партии товара C — цена единицы товара N — количество товара

Список примеров легко продлить, но и так ясно: простейшая модель, выраженная формулой умножения, весьма востребована. Эта *универсальность* характерна почти для всей математики. Существуют, разумеется, и очень специфические модели (например, математическая модель ядерного реактора), но и они строятся на базе тех же простых универсальных моделей.

Отметим, также, *обратимость* математических моделей. Так, из формулы, связывающей путь, скорость и время, можно через простейшие преобразования определить любой из параметров по двум остальным.

1.4. На троих

Теперь мы близки к тому, чтобы указать место программиста в новой, третьей кибернетической революции. Рассмотрим упрощённую схему разработки какой-либо компьютеризированной системы. Ею может быть и сложная система управления самолётом, и сравнительно простая бухгалтерская программа.



Разработку любой такой системы начинают с *постановки задачи*, — за это отвечают специалисты в данной области. Они должны чётко и однозначно описать исходные данные и все требования к будущей системе. Затем эстафету принимают математики, которые ищут методы решения поставленных задач, подбирают готовые алгоритмы, или же строят новые, и доказывают их правильность. Одним словом, строят математическую *модель*. Программистам же остаётся перенести эту модель с бумаги в электронные «мозги» компьютера.

На первый взгляд всё просто и ясно: здесь каждый делает своё дело. Но жизнь богаче и сложнее схем. Хорошо, если постановщик задачи может создать и математическую модель. В других ситуациях роль математика берёт на себя программист. Когда же роли разделяются (как на схеме), на стыках возможны трения: специалист затрудняется сформулировать задачу, математик — не всегда понимает специалиста, а программист — математика (тут сказывается уровень его математической подготовки). Как бы то ни было, конечным продуктом является система. Если она работает не так, как надо, следует «разбор полётов». В ходе

разбирательства могут исправляться и *программа*, и *математическая модель*, и даже *постановка задачи*.

Итак, тесная связь программиста с математиком очевидна. Программисту надо, по меньшей мере, владеть языком математики и понимать выражаемые им идеи. Именно это понимание, а не владение модным языком программирования, будет фундаментом и будущих ваших успехов, и профессионального долголетия.

1.5. Математика и математики

Раз уж мы коснулись математики, припомним школу. Основу математики составляют три сущности: аксиомы, теоремы и доказательства. *Аксиома* — это базовое утверждение, принимаемое без доказательств. *Доказательство* — это цепочка рассуждений, подчинённых нескольким простым правилам логики. Наконец, *теорема* — это утверждение, истинность которого вытекает из аксиом и доказана цепочкой рассуждений. В этом вся суть математики.

Приступая к очередной задаче, математик, прежде всего, включает интуицию и угадывает конечное или промежуточное решение. Как угадывает? — загадка. Но заподозрив решение, он должен показать его истинность, доказав одну или несколько теорем. Пока решение не доказано, это лишь догадка, *гипотеза*, а не теорема. Порой гипотеза бывает интуитивно понятной, но трудно доказуемой, некоторые доказательства даются лишь потомкам спустя десятки, сотни лет. Но, обретя доказательство, математик «умывает руки» — его работа сделана.

А что же программист? Обязан ли он доказывать теоремы? Если он изобрёл что-то новое, то придётся. Но чаще программист берёт готовые плоды математических усилий, и вправе освободить себя от перепроверки доказательств, — в конце концов, каждый грызёт свою морковку. Программисту важнее сосредоточиться на том, суть чего выражается в следующих вопросах:

- Какую реальность описывает данная математическая модель?
- Что за идеи лежат в основе решения?
- Как эффективно организовать данные и скомбинировать процедуры, чтобы превратить эту математическую модель в работающую программу?

К счастью, в отличие от изучающих математику инженеров других специальностей, у программиста есть возможность превращать изучаемый материал в работающие программы. Создание программ, оживляющих сухую математику, — прекрасный способ и освоения математики, и обретения опыта программирования, — *это наш путь*.

1.6. Дискретная математика

Математика включает много ветвей, крупных и мелких ответвлений. То, что применяют в инженерном деле и других науках, называют *прикладной* математикой. В ней, в свою очередь, выделяется ветвь, вплотную примыкающая к

компьютерной науке — информатике. Речь о *дискретной* математике, грань между которой и информатикой трудно различима. Но и дискретная математика весьма обширна, и потому в этой книге коснёмся лишь двух её разделов, на мой взгляд, самых интересных и полезных для программиста: *теории множеств* и *графов*.

У математики особый язык, и чтение математических книг, рекомендованных мною в списке литературы, предполагает владение оным. Но здесь я не буду «грузить» читателя чистой математикой, ведь эта книга для программистов. Не ищите здесь милых математикам крючков и загогулинок, — обойдёмся без них. Всё, что надо, я объясню «на пальцах». Мы не будем доказывать теорем, зато напишем море программ. Так будут убиты два зайца, которых мне не жаль: вы освоите много интересных и важных алгоритмов, а заодно научитесь создавать весьма сложные программы. В путь!

В конце каждой главы подводятся краткие итоги, а также даётся список рекомендуемой литературы. В этом списке особо отмечаются книги по теме текущей главы.

1.7. Итоги

1.7.1. В ходе эволюции живых организмов природа создала биологические кибернетические системы. Главное назначение этих систем — предсказание будущего с целью повышения конкурентных возможностей индивидуума.

1.7.2. Развитие человека и общества породило второй уровень кибернетических систем — символические модели, знакомые нам в виде наук и искусств. Эти системы тоже обладают предсказательной силой, и служат тем же целям: росту конкурентных преимуществ отдельных людей, обществ, стран, и человечества в целом.

1.7.3. Третий этап развития кибернетических систем связан с внедрением компьютеров: символические модели переводятся в электронную форму, и в этой работе не обойтись без программиста.

1.7.4. Главное конкурентное преимущество хорошего программиста заключено в способности воспринимать, анализировать, и превращать в работающие программы сложные символические модели, описывающие явления окружающего мира. Поскольку эти модели изложены языком математики, степень владения этой наукой определяет и квалификацию, и профессиональное долголетие программиста.

1.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
√ 11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 2
Объекты, множества и отношения

Дискретная математика манипулирует с неделимыми, то есть, *дискретными* объектами, — обычно это символы или числа, хотя ими наука не ограничена. Здесь мы бегло ознакомимся с понятиями, которые являются предметом этой книги. Наш обзор не претендует на полноту и строгость, — то и другое вы найдёте в предлагаемой литературе.

2.1. Понятия и объекты

Трудно сформулировать простые, интуитивно понятные вещи. Что такое *неделимый объект* — предмет манипуляций дискретной математики? Ответ кроется где-то внутри нас: поставьте перед собой фотографию или картину. Если это не художество абстракциониста, вы найдёте там нечто знакомое: людей, животных деревья, цветы. Но перед вами лишь клочок крашеной бумаги, никаких предметов там нет! Более того: всё, что мы видим, — это лишь матрица «пикселей» на сетчатке глаза. Однако наш мозг без труда выделяет в этом море пикселей знакомые предметы, запросто отличая кошку от собаки (кстати, невероятно трудная для компьютера задача).

Эта чудесная способность выработана годами эволюции, — в мозгу сформировались *понятия*, полезные для выживания организма. Сложились *понятия* не только о предметах в целом, но и о частях предметов: «плавник», «усы», «хвост». Более того, возникли и нематериальные *понятия*. Таких предметов, как «покой», «шутка», «родство», «мысль», в природе не существует, но мы представляем себе, что это такое, имеем *понятия* о них. А что выражают прилагательные и глаголы? Ими тоже передаются *понятия*, и вообще, всё, что мы ощущаем как *нечто, отличное от другого*, является *понятием*. Однако понятия, выражаемые существительными, нам особо дороги, мы назовём их *объектами*.

2.2. Дискретные объекты

Вот вам три предмета: вода, воздух, супчик, — это *вещества*, такие объекты характерны отсутствием определённой формы, а также способностью оставаться тем же самым при разделении на части. Вода остаётся водой хоть в реке, хоть в кружке, хоть в пригоршне.

А есть предметы иного рода: форма их относительно неизменна, и возможность деления отсутствует (разделённый на части объект прекращает существование). Так, мяч после серии ударов остаётся всё тем же узнаваемым мячом. Но нельзя, разрезав, сделать из него два или три мяча. То же будет и с кошкой. Такие объекты называют *дискретными*.

Неделимы, *дискретны* и многие нематериальные объекты: не бывает кусочков теоремы или ломтиков настроения. Далее мы будем работать лишь с

такими неделимыми *дискретными объектами*, как материальными, так и нематериальными.

2.3. Различимость и свойства

Любые два объекта *различимы*, *уникальны*, иначе это был бы один и тот же объект. Часто они различимы ввиду отличия их естественных *свойств*, как то: цвета, размера, веса и т.п. Абсолютно одинаковых объектов нет, но даже предметы, которые кажутся таковыми (шарики подшипника), различимы по своему месту в пространстве. Иногда такие предметы мы снабжаем дополнительными, искусственно введёнными свойствами. Например, сходящие с конвейера изделия снабжают уникальными заводскими номерами.

2.4. Простое и сложное

Дискретные объекты по определению неделимы, элементарны, однако при этом могут обладать сложной внутренней структурой. Здесь нет противоречия, поскольку при разрушении структуры объект перестаёт быть самим собой. Не зря говорят: из песни слова не выкинешь. Потому один и тот же объект может трактоваться и как *элементарный*, неделимый, и как сложная составная *система*. Например, участники дорожного движения склонны считать автомобили элементарными объектами, но в ремонтной мастерской преобладает другая точка зрения. Или возьмём город. Даже небольшой посёлок обладает весьма сложной структурой, включающей дома, коммуникации, дороги. Но на глобусе всё это обозначают лишь кружком — элементарным объектом.

Эта двойственность объектов никогда не затрудняла математиков, наделённых правом всё обозначать буквами, но озадачивала программистов, оперирующих со сложными структурами данных. К счастью, современные объектные технологии сняли остроту этой проблемы.

2.5. Абстрактное и конкретное

Сколько самых разных понятий роится в нашей голове? Не будь они как-то упорядочены, разложены «по полочкам», мозг погряз бы в этой информационной каше. И над этим тоже потрудилась эволюция, породив в мозгу *абстрактные* понятия о *конкретных* объектах. Конкретные объекты мы наблюдаем наяву: вот урчит у ног наша серая кошечка, а там крадётся соседский рыжий кот.

В борьбе за выживание побеждали те, кто быстрее и точнее реагировал на события, и потому мозг стал выделять в объектах или ситуациях наиболее *существенные* для принятия решения признаки. И теперь всех на свете *конкретных* кошек мозг представляет моделью *абстрактной* кошки вообще. Эта абстрактная кошка обожает молоко, гоняет птиц, а порой и мышей ловит, — всё это тотчас представляется нам при слове «кошка». Чья она, какой породы и масти? — этими менее существенными деталями мы интересуемся во вторую очередь.

На абстрактных понятиях первого уровня эволюция не успокоилась. Из кошек, собак, лошадей и прочей живности составилось более абстрактное понятие «животные». Последующее абстрагирование — порождение из частных понятий более общих — ведёт нас к вершине этой иерархии понятий — *абстрактному объекту*, с которого мы и начали главу. Стало быть, *абстракция* — отбрасывание несущественных признаков — природный процесс, совершенствовавший мозг.

2.6. Классификация и множества

Итак, своему биологическому выживанию мы обязаны естественной, природной абстракции. Но с тех пор, как люди занялись наукой и стали строить символические модели, пошёл и обратный процесс — *конкретизация*. Иначе говоря, перенося «на бумагу» то, что представлено в наших головах, мы стали раскладывать это по полочкам, *классифицировать*. Так возникло понятие *множества*, которое немецкий математик Кантор определил как *объединение в одно целое объектов, хорошо различимых нашей интуицией и нашей мыслью*.

2.6.1. Универсум

Определение Кантора позволяет «бросить в один мешок» всё, что мы можем себе представить, все конкретные и абстрактные объекты. Множество, содержащее все мыслимые объекты, называется *универсумом*. На первый взгляд такое множество кажется абсолютно бесполезным. Да, но зато универсум является тем резервуаром, из которого черпаются объекты для всех прочих множеств. Другими словами, все сколь-нибудь полезные множества являются подмножествами универсума.

Универсум единственен, но иногда это понятие намерено сужают. Например, при классификации всего живого, биологи не рассматривают неживые объекты, и для них универсум — это все живые организмы.

2.6.2. Предикаты

Объекты, входящие в полезные множества, выделяют из универсума путём применения *предикатов* — утверждений об объектах, которые могут оказаться либо истинными, либо ложными. Предикату можно сопоставить булеву функцию в языке **Pascal**. Так, например, для выделения из универсума всех живых объектов можно вообразить такую функцию:

```
function ОбменВеществ(объект): boolean;
```

Функция возвращает истину, если объекту-аргументу свойственен обмен веществ. Тогда сформировать множество всего живого можно перебором всех объектов универсума с применением условного оператора:

```
If ОбменВеществ(объект)
  then МножествоЖивого:= МножествоЖивого + объект
```

Действие предикатов обычно основано на свойствах объектов.

2.6.3. Эквивалентность

Об объектах одного множества говорят, что они *эквивалентны* в некотором смысле. Так, например, множество предметов белого цвета составляют белые объекты, — они эквивалентны в смысле одинаковой окраски. Множество чётных чисел содержит те, что при делении на два дают в остатке ноль — они эквивалентны в этом смысле. Другими словами, любому *множеству* соответствует некий предикат, и все его объекты *эквивалентны* в смысле этого *предиката*.

2.6.4. Выделение подмножеств

Классификация состоит в разбиении множества на менее крупные *подмножества*. Для этого нужны дополнительные предикаты: каждый разбивает исходное множество ровно на два непересекающихся подмножества: в первое попадают объекты, для которых предикат истинен, а во второе — для которых он ложен.

Разобьём множество всех автомобилей по цвету: белые, чёрные и красные (рис. 2-1). Здесь нужны три булевы функции, подобные той, что представлена выше (нужны три предиката). Применение ко всем автомобилям предиката **Белый** разобьет множество на «белые» и «не белые». Второй предикат разобьёт «не белые» на «чёрные» и «не чёрные». Предикат **Красный** разбивает подмножество «не чёрные» на «красные» и «не красные». Последнее подмножество можно подвергнуть дальнейшему разбиению.

Одно и то же множество может быть разбито разными предикатами на несколько разных подмножеств, как пересекающихся, так и не пересекающихся. Так, например, все автомобили могут быть классифицированы как по своему назначению (легковые, грузовые, автобусы и т.д.), так и по цвету. И тогда подмножество белых автомобилей наверняка пересечётся с подмножеством легковых.

Рис. 2-1 — Разбиение множества автомобилей тремя предикатами

2.6.5. Множества как объекты

Напомню, что согласно Кантору элементами множеств могут быть любые объекты. Но ведь само *множество* — это тоже *объект*, стало быть, множества могут входить в другие множества в качестве элементарных объектов. А множества множеств, в свою очередь, тоже можно трактовать как элементарные объекты и включать в другие множества, и т.д. Скоро и нам пригодятся эти трюки.

2.6.6. Операции с множествами

Если бы множества славились только своею универсальностью, «всеядностью», теория множеств не царствовала бы в математике. Силу ей даёт возможность *оперировать* с множествами. Важнейшими из этих операций являются *объединение* и *вычитание* множеств. На них строится ряд других операций, а также море полезных алгоритмов.

2.7. Отношения и графы

Итак, мы выяснили, что множества — один из тех инструментов, посредством которых человек строит символические модели, исследуя и познавая мир. Отметим, что классификация объектов, применяемая при построении множеств, основана лишь на *свойствах* объектов. Но с расширением круга решаемых задач всё трудней моделировать природу, опираясь лишь на свойства объектов. Многие явления объясняются не столько *свойствами*, сколько *отношениями* между объектами, их взаимными *связями*. Так, к примеру, поведение людей часто определяется не сколько их личными качествами, сколько отношениями в обществе (начальник-подчинённый, учитель-ученик). Понимание важности отношений породило, в конце концов, *теорию графов*.

2.7.1. Бинарные отношения

Бинарное отношение отражает некую связь между *двумя* объектами. Связи могут быть как зримыми, осязаемыми (дороги, провода), так и мыслимыми, воображаемыми (отношения подчинённости или родства). Отношения возможны только в паре, но иногда роль «напарника» исполняет тот же самый объект, — такое отношение называют рефлексивным. К примеру, любой человек является родственником самому себе.

Пара объектов может быть связана сразу несколькими отношениями разного рода. Например, два человека могут быть одновременно: а) родственниками, б) сослуживцами и в) соседями по даче. В теории графов рассматривают обычно лишь однородные отношения, когда между двумя объектами существует не более одной связи, и соответствующие графы называют *простыми*. Простые графы могут обладать самой причудливой «геометрией»: объекты могут быть связаны с одним или несколькими другими объектами, либо сами с собой, а то и вообще не связаны с другими объектами (изолированы).

2.7.2. Свойства отношений

Отношения характеризуют тремя свойствами: *рефлексивностью*, *симметричностью* и *транзитивностью*. Отношение может либо обладать, либо не обладать любым из этих свойств, что порождает несколько типов отношений.

Рефлексивность подразумевает возможность связи объекта с самим собой. Так, например, отношение «родственник» рефлексивно (каждый сам себе родня), а отношение «родитель» — нет (никто не рождает сам себя).

Симметричность отношения предполагает его взаимность. Так, отношение «одноклассники» симметрично, а отношение «родитель» — нет (сын не может быть родителем отца).

Транзитивным называют отношение, которое можно перенести со второго объекта на третий, четвёртый и последующие объекты. Так, отношение «родитель» не транзитивно: если некто родил сына, а этот сын — внука, то это не значит, что дед родил внука. Однако отношение «предок» транзитивно: и отец, и деды, и все прадеды являются предками человека.

Рефлексивные и симметричные отношения отмечают на графах линиями и стрелками так, как показано на рис. 2-2.

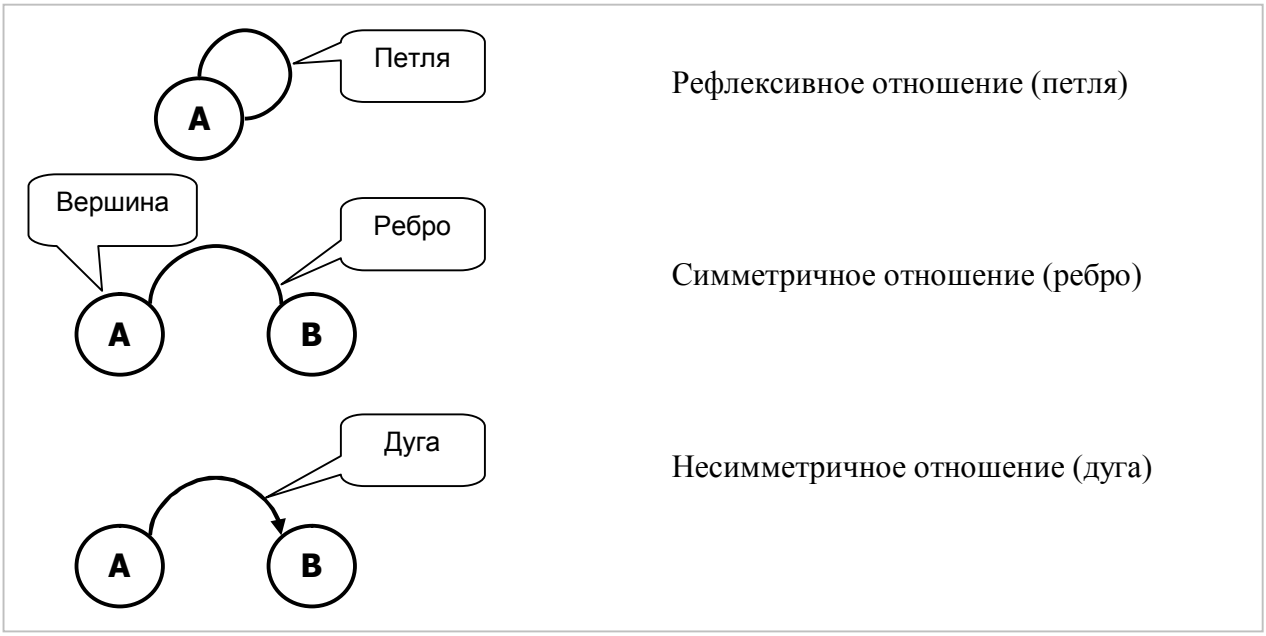


Рис. 2-2 — Графические обозначения отношений

2.7.3. Связи — это тоже объекты

Итак, на рис. 2-2 показаны примеры графов. Объекты здесь изображены кружками, их называют *вершинами* или *узлами* графа. Вершины содержат в себе некую информацию об объектах, например, название города, его населённость и т.п. Связи между вершинами обозначают *дугами* (со стрелкой) или *рёбрами*, они тоже могут содержать информацию, например, расстояние между городами.

Отметим, что объектами являются не только вершины графа, но и связи между ними (дуги, рёбра). Стало быть, и вершины, и связи можно сгруппировать в два множества: множество вершин X, и множество связей A. Тогда граф G в целом будет множеством, состоящим из двух элементов-множеств, что обычно записывают так: $G = \langle X, A \rangle$.

2.8. Объекты, объекты, объекты...

Итак, в начале этой главы мы выяснили, что значит элементарный дискретный объект. Затем узнали, что в ходе построения математических моделей объекты объединяются в множества, а множества сами по себе тоже можно трактовать как дискретные объекты. Далее уяснили, что многие модели могут быть построены лишь с учётом связей между объектами, причём эти связи тоже являются объектами. В конце концов, граф в целом состоит из двух множеств: множества вершин и множества связей, и тоже является объектом. Всё это наводит на мысль сконструировать *иерархию объектов*, способную вместить в себя всё богатство этого понятия — *объект*. Эту задачу начнём решать, начиная со следующей главы.

2.9. Итоги

2.9.1. В ходе эволюции в мозгу человека сформировались *понятия*, — мы ощущаем их как нечто, отличное от другого.

2.9.2. Среди понятий мы выделяем неделимые *дискретные* объекты, к которым относятся и реальные предметы, и части предметов, и абстрактные понятия, не встречаемые в природе (такие, как «мысль», «настроение», «теорема»).

2.9.3. Дискретный объект нельзя разделить, не разрушив. И в то же время он может обладать внутренней структурой и состоять из других дискретных объектов.

2.9.4. Дискретные объекты обладают рядом свойств, и на основе этих свойств могут группироваться в *множества* и *подмножества* (посредством предикатов). Операции с множествами составляют основу современной математики.

2.9.5. При построении моделей, отражающих реальность, важно учитывать не только свойства объектов, но также их *отношения*. Отношения в математике изображают посредством *графов*.

2.10. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
√ 11	Турчин В.Ф.	Феномен Науки	
√ 12	Хаггарти Р.	Дискретная математика для программистов	

Глава 3
Объекты в языке Delphi

Задача программиста — перенос моделей из дискретной математики в «мозг» компьютера. Насколько легко мы справимся с этой задачей, отчасти определяется выбранным языком. Многим дискретным объектам присуща двойственность: с одной стороны они трактуются как элементарные, а с другой — представляют собой сложные структуры данных (массивы, множества или даже графы). Когда-то эта проблема решалась сложными окольными путями, но с появлением объектно-ориентированных языков всё изменилось: объекты, словно нарочно, придуманы именно для этой цели.

Мы воспользуемся языком **Delphi**, известным по одноимённой среде программирования. **Delphi** — это развитие языка **Object Pascal**, а тот, в свою очередь, порождён от **Borland Pascal**, хорошо знакомого многим студентам и школьникам. Надеюсь, это облегчит усвоение представленных здесь решений. Отметим, что во многих книгах при объяснении алгоритмов тоже предпочитают паскале-подобный синтаксис ввиду его простоты и выразительности.

Хотя язык **Delphi** в своём развитии ушёл далеко от базового Паскаля, для восприятия книги не требуется досконально владеть его тонкостями, — хватит основ объектно-ориентированного программирования (ООП), приобретённых в рамках **Borland Pascal**. Наиболее существенные новшества и отличия от **Borland Pascal** будут пояснены в этой главе. Тем, кто не владеет основами ООП, рекомендую сначала обратиться к предлагаемой литературе, а если вы хорошо знакомы с **Delphi**, можете без ущерба пропустить эту главу.

3.1. Классы и объекты

Начнём с синтаксического новшества в объявлении объектов: если в **Borland Pascal** для этого применялось ключевое слово **Object**, то в **Delphi** для этой цели предусмотрено ключевое слово **Class**, например:

```
type   T1 = class(TObject)   end;
```

Здесь объявлен объектный тип данных **T1**, произведённый от предопределённого в языке типа данных **TObject** (в данном случае он совпадает с предком). Таким образом, в языке чётко разделены два понятия: *класс* — это *тип* данных, а *объект* — это *переменная* объектного типа.

3.2. Объекты – это динамические переменные

Следующая отличие **Delphi** состоит в том, что переменные объектного типа здесь являются *указателями* на динамические переменные, — статических объектов в **Delphi** не предусмотрено. Таким образом, для работы с объектами

недостаточно одного лишь объявления: надо ещё создать объект конструктором, а впоследствии удалить из динамической памяти, например:

```
var X: TObject;  // объявление
begin
    X:= TObject.Create;   // создание объекта
    { работа с объектом... }
    X.Free;               // освобождение объекта
end.
```

3.2.1. Разыменование по умолчанию

Поскольку компилятору языка известно о динамической природе объектов, для доступа к полям и методам он не требует разыменования таких переменных стрелочкой «^». Это существенно разгружает тексты программ и облегчает их восприятие.

3.2.2. Конструкторы

Как видно из примера выше, конструктор объекта — это *функция*, возвращающая указатель на объект. Вызов конструктора предваряется префиксом в виде имени класса, к которому принадлежит создаваемый объект. То есть, общий синтаксис вызова конструктора таков:

```
Объект:= ИмяКласса.ИмяКонструктора
```

В корневом классе **TObject** предусмотрен конструктор по имени **Create** (создать), он не требует параметров. Однако классы-наследники могут снабжаться разными конструкторами: с различными названиями, с параметрами, и даже несколькими конструкторами, например:

```
type  TMyClass = class(TObject)
        m1: integer;
        constructor Create(aVal: integer);
        constructor InitOne;
        constructor InitTwo;
      end;

constructor TMyClass.Create(aVal: integer);
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= aVal;
end;

constructor TMyClass.InitOne;
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= 1;
end;

constructor TMyClass.InitTwo;
begin
  inherited Create;  // обязательный вызов унаследованного конструктора
  m1:= 2;
```

```
end;

var X0, X1, X2 : TMyClass;

begin
  X0:= TMyClass.Create(10);
  X1:= TMyClass.InitOne;
  X2:= TMyClass.InitTwo;
  ...
end.
```

Как показывает этот пример, внутри нового конструктора вначале обязательно вызывают унаследованный конструктор. Хотя конструктор является функцией, её тип в объявлении не указывают (тип очевиден), и возвращаемое значение в теле не определяют.

3.2.3. Инициализация полей

Назначение конструктора — создать объект и заполнить его поля (инициализировать). Корневой конструктор **`Create`** только очищает все поля объекта, — иногда это всё, что требуется. После очистки, поля объекта, в зависимости от их типа, принимают следующие значения:

Тип поля	Значение
Число	**0**
Булево	**false**
Символ	**#0**
Строка	**Пустая**
Указатель	**nil**

Таким образом, унаследованным классам не всегда нужен новый конструктор, он необходим лишь для инициализации полей, содержащих непустые значения.

3.2.4. Деструкторы

Назначение деструктора — освободить память, занимаемую объектом. Деструктор корневого класса **`TObject`** объявлен как виртуальный:

```
    destructor Destroy; virtual;
```

Его реализация пуста, сам по себе он ничего не делает, зато позволяет переопределять себя в наследниках. Переопределение нужно, если поля наследника тоже являются объектами, и тогда деструктор прежде уничтожает эти поля, а затем вызывает унаследованный деструктор. Запомните: унаследованный конструктор всегда вызывают *первым*, а деструктор — *последним*, например:

```
type  TMyClass = class(TObject)
        mObj: TObject;
        constructor Create;
        destructor Destroy; virtual;
```

```
      end;

constructor TMyClass.Create(aVal: integer);
begin
  inherited Create;        // унаследованный конструктор
  mObj:= TObject.Create;   // инициализация поля
end;

destructor TMyClass.Destroy;
begin
  mObj.Free;          // освобождение поля
  inherited Destroy; // унаследованный деструктор
end;

var X: TMyClass;

begin
  X:= TMyClass.Create;
  { работа с объектом }
  X.Free;  // неявный вызов деструктора Destroy
end.
```

Но почему в этом примере вызван не деструктор **Destroy**, а процедура **Free**? Посмотрите, как реализован метод **Free**:

```
procedure TObject.Free;
begin
  if Self <> nil then Destroy;
end;
```

Перед вызовом деструктора метод проверяет, не вызван ли он для несуществующего объекта (здесь **Self** — это указатель на данный объект).

3.2.5. Очистка переменных-классов

Хотя при удалении объекта деструктор и освобождает занимаемую им память, он не трогает сам указатель на объект (переменную), например:

```
var X: TObject;

begin
    Writeln('1= ', Assigned(X));   // false
    X:= TObject.Create;
    Writeln('2= ', Assigned(X));   // true
    X.Free;
    Writeln('3= ', Assigned(X));   // true
    X:=nil;
    Writeln('4= ', Assigned(X));   // false
    Readln;
end.
```

Когда надо отметить факт уничтожения объекта, освободившейся переменной принудительно присваивают пустое значение **NIL**.

3.3. Методы класса

Методы класса — ещё одно ценное средство **Delphi**. Они могут вызываться двояко: 1) как обычно — с указанием префикса-переменной, 2) указанием в качестве префикса имени типа (как при вызове конструктора). Иначе говоря, для вызова этих методов наличие объектов не обязательно. Методы, встроенные в корневой класс **TObject**, дают ряд полезных возможностей по отслеживанию типов переменных и их размеров на этапе исполнения программы. Рассмотрим некоторые из них, а именно:

```
TObject = class
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    ...
  end;
```

3.3.1. Определение имени класса

Метод **ClassName** возвращает строку с именем класса, совпадающую с именем в тексте программы с точностью до регистра букв. Для сравнения имени класса со строкой, или с именем другого класса лучше подходит функция **ClassNameIs**, игнорирующая регистр букв, например:

```
var T1, T2 : TObject;

begin
  Writeln(TObject.ClassName);               // 'TObject' - как в исходном коде
  Writeln(TObject.ClassName='TOBJECT');     // false (несовпадение регистра)
  Writeln(TObject.ClassNameIs('TOBJECT')); // true (регистр игнорируется)
  Writeln(TObject.ClassNameIs('tobject')); // true (регистр игнорируется)
  Writeln(TObject.ClassNameIs('obj'));      // false
  T1:= TObject.Create;
  T2:= TObject.Create;
  Writeln(T1.ClassNameIs(T2.ClassName));    // true
  Readln;
end.
```

3.3.2. Сравнение типов

Сравнивать типы объектов можно функциями **ClassType** и **InheritsFrom**. Первая из них возвращает тип **TClass**, который, по сути, является *указателем* на специальную структуру данных, связанную с классом: для переменных одного класса эти указатели совпадают.

Метод **InheritsFrom** действует тоньше, возвращая **true**, если данный класс, либо совпадает с классом-аргументом, либо унаследован от него, например:

```
type

  T1 = class(TObject)  // унаследован от корневого
    m1: integer;
  end;

  T2 = class(T1)  // унаследован от T1
    m2: integer;
  end;

var V0 : TObject;
    V1 : T1;
    V2 : T2;

begin
    V0:= TObject.Create;
    V1:= T1.Create;
    V2:= T2.Create;

    Writeln(V0.ClassType = TObject);  // true
    Writeln(V1.ClassType = TObject);  // false
    Writeln(V2.ClassType = TObject);  // false

    Writeln(V0.InheritsFrom(TObject));  // true
    Writeln(V1.InheritsFrom(TObject));  // true
    Writeln(V2.InheritsFrom(TObject));  // true

    Writeln(V0.InheritsFrom(T1));  // false
    Writeln(V1.InheritsFrom(T1));  // true
    Writeln(V2.InheritsFrom(T1));  // true

    Writeln(V0.InheritsFrom(T2));  // false
    Writeln(V1.InheritsFrom(T2));  // false
    Writeln(V2.InheritsFrom(T2));  // true
    Readln;
end.
```

3.3.3. Сравнение типов операцией is

На практике типы сравнивают основанной на функции **InheritsFrom** операцией **is**, например:

```
type

  T1 = class(TObject)
    m1: integer;
  end;

  T2 = class(T1)
    m2: integer;
  end;

var V0 : TObject;
    V1 : T1;
    V2 : T2;

begin
    V0:= TObject.Create;
    V1:= T1.Create;
```

```
    V2:= T2.Create;

    Writeln(V0 is TObject);  // true
    Writeln(V1 is TObject);  // true
    Writeln(V2 is TObject);  // true

    Writeln(V0 is T1);  // false
    Writeln(V1 is T1);  // true
    Writeln(V2 is T1);  // true

    Writeln(V0 is T2);  // false
    Writeln(V1 is T2);  // false
    Writeln(V2 is T2);  // true

    Writeln(V1 is V0.ClassType);  // true
    Writeln(V1 is V2.ClassType);  // false

    Readln;
end.
```

3.4. Копирование объектов

Оператор присваивания копирует не объекты, а указатели на них. При этом компилятор требует, чтобы справа от знака присваивания располагался либо объект того же класса, что и слева, либо его наследник, например:

```
  // определения см. в предыдущем примере
  v0:= v1;  // верно
  v1:= v0;  // ошибка
```

Это общее правило основано на полиморфизме в ООП, оно гарантирует, что в случае необходимости будет вызван метод того объекта, на который указывает переменная в текущий момент, будь то объект того же типа, что и переменная, либо любой его наследник. Это срабатывает, когда вызываемый метод определён в родительском классе. Если же надо обратиться к новым, не существующим в предке полям и методам, прибегают к приведению типов.

3.5. Приведение типов операцией as

Приведение типа не влечёт преобразований данных. Оно лишь заставляет компилятор трактовать некую переменную не так, как она объявлена, а так, как надо программисту в данном операторе. При этом ответственность за возможную ошибку переходит от компилятора к программисту.

К примеру, некоторое число **N** можно трактовать как код символа: **Char(N)**. Точно так же можно приводить и объектные переменные, трактуя предка, как потомка. Но если в первом случае мы ничем не рискуем, то в случае с объектами наша невнимательность может повлечь крах или необъяснимое поведение программы. Поэтому для объектных типов в **Delphi** предусмотрена особая конструкция приведения типов ключевым словом **as**. Рассмотрим следующий пример:

```
type

  T1 = class(TObject)   m1: integer;  end;
  T2 = class(T1)        m2: integer;  end;
  T3 = class(T2)        m3: integer;  end;

var V1 : T1;

begin
    V1:= T1.Create;   // "Родной" тип данных
    Writeln(V1.m1);   // Всё верно
    V1.Free;

    V1:= T2.Create;   // Тип потомка
    // Корректное приведение типа:
    Writeln(T2(V1).m2);         // Жёсткое приведение типа
    Writeln((V1 as T2).m2);     // Мягкое приведение типа
    // Некорректное приведение типа:
    Writeln(T3(V1).m3);         // Здесь ошибка приведения не обнаружена
    Writeln((V1 as T3).m3);     // Обнаружена ошибка EInvalidCast
end.
```

Здесь представлена иерархия из трёх классов: **T1 -> T2 -> T3**. Переменная **V1** имеет тип родителя, что даёт нам право присваивать ей указатели на потомков без приведения типов. Однако для обращения к новым полям потомка мы должны привести переменную к типу этого потомка. Традиционное «жёсткое» приведение типа бесконтрольно и может повлечь труднообъяснимые ошибки. «Мягкое» приведение операцией **as** порождает дополнительный проверяющий код, который генерирует исключение **EInvalidCast** в случае несоответствия типов при исполнении программы. Оно же побуждает проверить соответствие типов в ходе компиляции.

Альтернативой может быть следующий код:

```
    if V1 is T3
      then {безопасно приводим V1 к типу T3}
      else {сообщаем об ошибке};
```

3.6. Размер объекта

Иногда требуется знать объём памяти, занимаемый объектом. Однако традиционная псевдо-функция **SizeOf**, применённая к объектной переменной, вернёт размер указателя, а это не то, что нужно. Задачу решает функция класса **InstanceSize**, которая возвращает реальный объём памяти, занимаемый полями объекта. Для корневого «пустого» объекта **TObject** функция возвращает размер указателя, поскольку такой указатель на специальную структуру данных, описывающую класс, скрыт где-то в недрах корневого объекта. Для всех унаследованных классов функция возвращает сумму размеров полей плюс размер указателя. Для классов, определённых в предыдущем примере, будут выданы следующие результаты:

```
    // Выводится размер указателя
    Writeln(SizeOf(T1));  // 4
    Writeln(SizeOf(T2));  // 4
    Writeln(SizeOf(T3));  // 4

    // Выводится размер объекта вместе со скрытым полем
    Writeln(TObject.InstanceSize);  // 4
    Writeln(T1.InstanceSize);       // 8
    Writeln(T2.InstanceSize);       // 12
    Writeln(T3.InstanceSize);       // 16
```

3.7. Области видимости

Области видимости полей и методов нужны для частичного сокрытия внутренностей объекта, что в целом повышает надёжность программ. Чтобы определить полю или методу область видимости, его объявление помещают в одну из четырёх секций класса, мы рассмотрим только три из них:

```
Type
  T = class(TObject)
    private
      // Приватная секция
      m1: integer;
      ...
    protected
      // Защищённая секция
      m2: integer;
      ...
    public
      // Общедоступная секция
      m3: integer;
      ...
  end;
```

Краткое описание областей видимости дано в табл. 3-1.

Табл. 3-1 — Описание областей видимости

Наименование секции	Ключевое слово	Описание
Приватная	**private**	**Поля и методы доступны только в модуле, где описан объект.**
Защищённая	**protected**	**Поля и методы доступны за пределами модуля, но только из методов данного объекта и его наследников.**
Общедоступная	**public**	**Поля и методы доступны в любом месте программы.**

Секции могут повторяться и чередоваться в любом порядке.

3.8. Свойства

Ещё одним ценным средством повышения надёжности программ являются *свойства* — механизм доступа к полям через специально созданные для этого методы. Хотя реализация этих методов обычно несложна, дополнительный код может ухудшить понимание программы. Это единственная причина того, что в

данной книге механизм свойств не используется, и обращение к полям объектов выполняется непосредственно.

3.9. Статические, динамические, виртуальные и абстрактные

Этими титулами награждаются методы объектов. Соответствующие ключевые слова и особенности методов представлены в табл. 3-2.

Табл. 3-2 — Разновидности методов

Метод	Ключевое слово	Описание метода
Статический	**Нет**	**Реализуют действия, выполняемые одинаково во всех потомках данного класса. Никаких ключевых слов в объявлении метода не используется.**

Виртуальный	**virtual**	**Реализует разное исполнение однотипных действий в разных потомках класса.**
		Объявление метода в родителе снабжается ключевым словом virtual:
		`procedure ABC; virtual;`
		Объявление метода в потомках снабжается ключевым словом override:
		`procedure ABC; virtual; override;`

Динамический	**dynamic**	**То же, что и виртуальный метод, но таблица динамических методов построена компактней, что экономит память.**

Абстрактный	**abstract**	**Методы, объявленные в родителе как абстрактные, не нуждаются в реализации, но и вызваны быть не могут. Эти методы обязательно переопределяются в потомках.**
		Пример объявления в родителе:
		`procedure ABC; virtual; abstract;`
		Пример объявления в потомках:
		`procedure ABC; virtual; override;`

Больше информации на эту тему найдёте в рекомендуемой литературе.

3.10. Прочие замечания

Теперь обращу ваше внимание на некоторые часто используемые в книге средства языка.

3.10.1. Локальные процедуры и функции

Сложные процедуры и функции вполне естественно надо разбивать на ряд простых, что и предусмотрено во всех языках программирования. В Паскале и **Delphi**, в отличие, скажем, от Си, можно вставлять такие вспомогательные процедуры и функции внутрь других процедур и функций (подобно матрёшкам), что делает программы более надёжными и обозримыми. Видимость локальных процедур и функций ограничена той процедурой или функцией, в которую она вставлена. В книге вы найдёте много примеров подобных решений.

3.10.2. Оператор whith

Оператор **whith** создаёт локальную область видимости полей записи или полей и методов класса, например:

```
type

  T0 = class(TObject)
    m1: integer;
    m2: integer;
    m3: integer;
  end;

var  V0 : T0;
begin
  ...
  // Вместо
  V0.m1:= 1;  V0.m2:= 2;  V0.m3:= 3;
  // Можно записать так:
  with V0 do begin
    m1:= 1;    m2:= 2;    m3:= 3;
  end;
  ...
end
```

Такие сокращения разгружают текст и облегчают понимание программы.

3.10.3. Квалификатор Self

Ключевое слово **Self** означает ссылку на текущий объект внутри методов класса, например:

```
function TNode.GenGammaOut: TSet;
begin
  ...
  // здесь текущий объект передаётся в качестве параметра в процедуру Insert
  Result.Insert(Self);
  ...
end;
```

3.10.4. Перегрузка операторов

Эта возможность языка состоит в том, что вызов функции заменяется одним из общепринятых математических знаков (+, -, * и т.д.). Так, к примеру, функцию вставки объекта в множество можно «перелицевать» в оператор сложения. Перегрузка операторов с одной стороны улучшает читабельность программы, а с другой — усложняет поиск ошибок, когда что-то идёт не так. И потому в обществе не сложилось единого мнения о пользе или вреде перегрузки. В данной книге перегрузка не используется.

3.11. Итоги

3.11.1. Объектные переменные **Delphi** отражают двойственную природу реальных дискретных объектов: с одной стороны они элементарны, а с другой — могут представлять собой сколь угодно сложные структуры данных.

3.11.2. В языке **Delphi** чётко различаются два понятия: *класс* — это объектный тип, а *объект* — переменная этого типа.

3.11.3. Все объектные переменные в **Delphi** являются указателями на динамические переменные. Помимо объявления, объекты надо также создавать и уничтожать.

3.11.4. Объектные переменные в **Delphi** разыменованы по умолчанию, стрелки «^» не обязательны.

3.11.5. При создании объекта конструктором все его поля автоматически очищаются.

3.11.6. Для контроля и приведения типов объектных переменных используют методы корневого класса **TObject**, а также операции **as** и **is**.

3.11.7. Реальный объём памяти, занимаемый объектом, возвращает метод класса **InstanceSize**.

3.11.8. Ограничение областей видимости полей и методов, а также применение свойств снижает риск непреднамеренного искажения данных и повышает надёжность программ.

3.12. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
√ 4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
√ 5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 4
Базовый объект

Начинаем воплощать нашу мечту — создавать универсальный объект, сочетающий, на первый взгляд, не сочетаемое: с одной стороны он будет элементарным, а с другой — может представлять сколь угодно сложную структуру данных. Современные объектные технологии дают нам такую возможность.

Разумеется, что одним объектным классом нам не обойтись: для решения последующих задач нам потребуется иерархия объектов. Забегая немного вперёд, посмотрим на рис. 4-1, где представлена эта иерархия. Все будущие классы будут порождены нами от базового класса **TItem** — *элемент*. Это значит, что такие сложные структуры, как очереди, стеки, множества, графы и прочие будут обладать свойствами элементарного объекта. При необходимости мы сможем, к примеру, создать множества, элементами которых будут другие множества, и помещать их в очередь или стек. Или создавать множества из графов, очередей, или создавать очереди очередей и т.д. Короче, мы забудем о любых ограничениях на сложность объектов.

4.1. Базовый класс

Поскольку классу **TItem** уготована роль предка всех прочих классов, в ходе проектирования важно определить наиболее общие методы, свойственные всем его потомкам. Практика показала, что в объектах-потомках потребуются три базовые возможности:

- возможность выводить информацию о себе в текстовый файл и на экран;
- возможность сравнивать себя с другим объектом;
- возможность порождать свою копию.

Необходимость отображать объект в текстовой форме очевидна, иначе как увидеть результаты программы? Возможность сравнивать объекты нужна при вставке их в множества, ведь дубликаты в множествах запрещены, а упорядочение приветствуется. Необходимость копирования объясняется тем, что оператор присваивания копирует только указатель на объект, но не сам объект, а это порой требуется. Всё это приводит к следующему объявлению базового класса (см. модуль **Root**):

```
TItem = class (TObject)
  public
    function Compare(arg: TItem): TCompare; virtual;  // сравнение
    function Copy: TItem;  virtual; abstract;         // копирование
    procedure Print(var aFile: TextFile); virtual; abstract; // вывод в файл
    procedure Expo;  // вывод на экран
end;
```

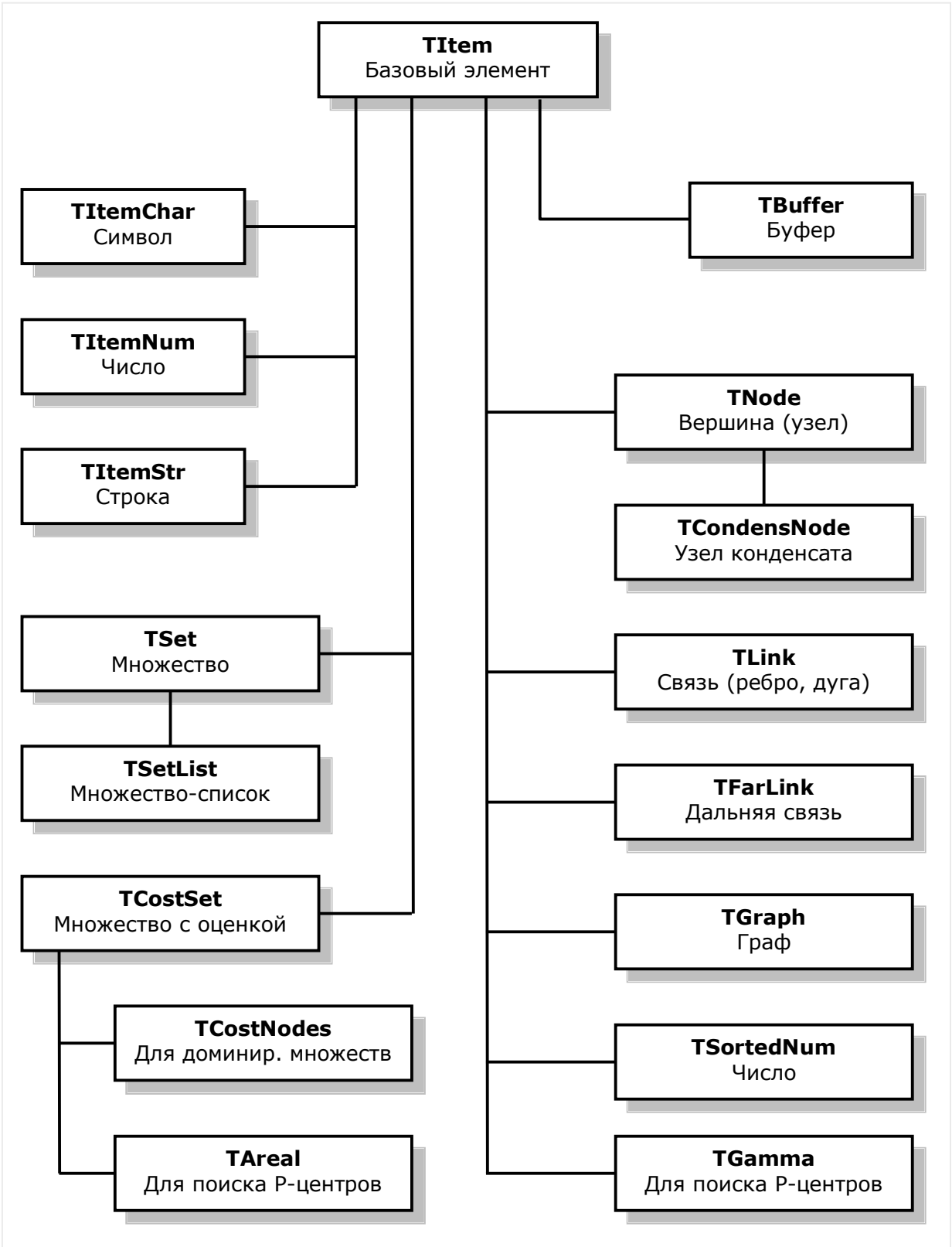


Рис. 4-1 — Схема наследования основных классов

4.1.1. Вывод в текстовой форме

Вывод в текстовый файл будем выполнять методом **Print**, принимающим один параметр — открытый текстовый файл. Кодировать метод в базовом классе пока не имеет смысла, потому он объявлен абстрактным.

Зато второй метод — **Expo**, предназначенный для вывода объекта на экран, может быть реализован сразу:

```
procedure TItem.Expo;        // Метод отображения на экране
begin
  Print(Output);    // Вывод на экран
end;
```

Метод **Expo** объявлен статическим, а это значит, что в будущем переопределять его мы не будем. Странно, что он вызывает нереализованный пока метод **Print**, и компилятор не возражает против этого. Но таков общий подход в ООП: если предку нужен какой-то метод, реализуемый лишь в потомках, мы объявляем его в предке абстрактным.

4.1.2. Сравнение объектов

Сравнивать объекты придётся очень часто, особенно работая с множествами. Как правило, интересующие нас объекты не являются числами, и, тем не менее, нам придётся их сравнивать на равенство/неравенство и на больше/меньше. С этой целью в модуле **Root** объявлено перечисление:

```
// Возможные результаты сравнения объектов

  TCompare = (cmpEq,      // объекты совпадают или эквивалентны
              cmpLess,    // меньше
              cmpGreate,  // больше
              cmpIncomp   // несравнимы (Incomparable)
             );
```

Метод **Compare** будет возвращать одно из этих значений. Здесь, в корневом предке, он реализован так:

```
function TItem.Compare(arg: TItem): TCompare;
begin
  if arg = Self
    then Result:= cmpEq       // совпадают
    else Result:= cmpIncomp;  // несравнимы
end;
```

В будущих потомках он будет, как правило, переопределяться.

4.1.3. Копирование объекта

Метод копирования **Copy** объявлен виртуальным и абстрактным, т.е. будет реализован в потомках.

4.2. Демонстрационные классы

Бедные возможности базового класса очевидны: ведь он не содержит никакой информации, и потому сам по себе ни на что не годен. Зато служит фундаментом для многих других классов, — мы продемонстрируем это на примере трёх реальных классов: *символ*, *число* и *строка* (см. модуль **Items**).

Класс **TItemChar** содержит одно информационное поле — символ. В нём уже реализованы методы, которые в предке были абстрактными.

Листинг 4-1 — Объявление и реализация класса *символ*

```
TItemChar = class (TItem)
    private
      mData: Char;
    public
      constructor Create(arg: Char);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: Char;
  end;

implementation

constructor TItemChar.Create(arg: Char);
begin
  inherited Create;
  mData:= arg;
end;

function TItemChar.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemChar) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemChar).mData = mData
    then Result:=cmpEq
    else if (arg as TItemChar).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate;
end;

function TItemChar.Copy: TItem;
begin
  Result:= TItemChar.Create(mData);
end;
```

```
procedure TItemChar.Print(var aFile: TextFile);
begin
  Write(aFile, mData:2)
end;

function TItemChar.GetData: Char;
begin
  Result:= mData
end;
```

Объекты **TItemNum** (число) и **TItemStr** (строка) реализованы аналогично, за подробностями обращайтесь к модулю **Items**. Ниже показано применение этих трёх объектов.

Листинг 4-2 — Демонстрация полиморфизма на примере трёх объектов

```
uses
  Root in '..\Common\Root.pas',
  Items in '..\Common\Items.pas';

var  Buf : array [1..3] of TItem;
       i : integer;
begin
    Buf[1]:= TItemChar.Create('A');      // символ
    Buf[2]:= TItemNum.Create(10);        // число
    Buf[3]:= TItemStr.Create('Class');   // строка
    for i:= 1 to 3 do begin
      Buf[i].Expo;
      Writeln;
      Buf[i].Free;
    end;
    Readln;
end.
```

Хотя элементы массива **Buf** относятся к типу **TItem**, программа помещает туда разнотипные элементы-потомки. Обычно массивы этого «не любят», но здесь всё в порядке, — сказывается полиморфизм ООП.

4.3. Итоги

4.3.1. Объектные технологии позволяют отразить в программных объектах противоречивую сущность объектов реальных: с одной стороны, такие объекты могут быть весьма сложными, с другой — рассматриваются как элементарные, неделимые.

4.3.2. Вся гамма дискретных объектов, используемых для решения последующих задач, будет представлена иерархией объектов, происходящих от корневого объекта класса **TItem** — «элемент».

4.3.3. Корневой прародитель всех элементов должен обладать только методами, свойственными всем его потомкам. Эти методы будут обеспечивать:

- вывод объекта в файл и на экран (**`Print`**, **`Expo`**);
- сравнение элементов (**`Compare`**);
- создание копии элемента (**`Copy`**).

4.4. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
√ 4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
√ 5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 5
Универсальный буфер

В этой главе мы разработаем первый полезный объект, который потребуется нам в последующих проектах — универсальный буфер.

5.1. Требования

Для размещения данных в оперативной памяти язык **Delphi** и сопутствующие библиотеки предлагают богатый арсенал средств: статические и динамические массивы, коллекции, списки. Но мы разработаем своё собственное вместилище элементов, обладающее нужными нам свойствами и возможностями, а именно:

- вместимость, ограниченную лишь объёмом памяти;
- возможность доступа к элементам по индексу, подобно массиву;
- вставку и извлечение элементов по принципу очереди;
- вставку и извлечение элементов по принципу стека;
- реверс элементов в буфере — перестановку их в обратном порядке;
- очистку буфера с уничтожением и без уничтожения элементов.

Класс универсального буфера произведём от нашего базового класса **TItem**, поэтому объект-буфер будет обладать всеми его возможностями.

5.2. Объявление

Соответствующее нашим намерениям объявление класса **TBuffer** будет таким (см. модуль **Items**):

```
TBuffer = class (TItem)
    private
      mCount: integer;                // счётчик элементов
      mHead, mQue: PStackRec;         // указатели на начало и конец буфера
      procedure Clr(aDestroy: boolean);
    public
      destructor Destroy;  override;
      function Copy: TItem; override; // создание копии
      procedure Put(arg: TItem);      // занесение в очередь
      function Get: TItem;            // извлечение из очереди
      procedure Push(arg: TItem);     // занесение в стек
      function Pop: TItem;            // извлечение из стека
      procedure Reversion;            // реверс буфера
      function GetByIndex(aIndex: integer): TItem; // доступ по индексу
      function GetCount: integer;     // количество элементов в буфере
      procedure Clear;                // очистка без уничтожения объектов
      procedure ClrAndDestroy;        // очистка с уничтожением объектов
      procedure Print(var aFile: TextFile); override; // вывод в файл
  end;
```

5.3. Реализация

Заявленный объект-буфер можно реализовать разными средствами. Мы применим односвязный список, достигая поставленных целей относительно просто. Класс **`TBuffer`** размещён в модуле **`Items`**, листинг его представлен ниже.

Листинг 5-1 — TBuffer -- класс для буфера, очереди и стека

```
// Вспомогательная структура для организации односвязного списка

PStackRec = ^TStackRec;
  TStackRec = record
    mItem : TItem;             // элемент списка
    mNext : PStackRec;         // ссылка на следующий
  end;

// Создание копии буфера из существующего

function TBuffer.Copy: TItem;
var t: TItem;
    i: integer;
begin
  Result:= TBuffer.Create;
  for i:= 1 to GetCount do begin
    t:= Get;
    Put(t);
    TBuffer(Result).Put(t);
  end;
end;

// Уничтожение буфера без уничтожения элементов

destructor TBuffer.Destroy;
begin
  Clear;
  inherited;
end;

// Очистка буфера

procedure TBuffer.Clr(aDestroy: boolean);
var i : integer;
    t : TItem;
begin
  for i:= 1 to mCount do begin
    t:= Get;
    if aDestroy then t.Free
  end;
end;

// Очистка с уничтожением элементов

procedure TBuffer.ClrAndDestroy;
begin
  Clr(true)
end;

// Очистка без уничтожения

procedure TBuffer.Clear;
```

```
begin
  Clr(false)
end;

// Получение счётчика элементов

function TBuffer.GetCount: integer;
begin
  Result:= mCount;
end;

// Push - помещение в стек (в начало буфера)

procedure TBuffer.Push(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= mHead;
  mHead:= r;
  Inc(mCount);
  // Обновляем указатель на конец списка
  if not Assigned(mQue) then mQue:= r;
end;

// Put - помещение в очередь (в конец буфера)

procedure TBuffer.Put(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= nil;
  // Присоединяем к концу списка
  if Assigned(mQue) then mQue^.mNext:= r;
  mQue:= r;
  Inc(mCount);
  // Обновляем указатель на начало
  if not Assigned(mHead) then mHead:= r;
end;

// Извлечение из очереди и стека (из начала списка )

function TBuffer.Get: TItem;
var r: PStackRec;
begin
  Result:= nil;
  if not Assigned(mHead) then Exit;
  r:= mHead;
  Result:= mHead^.mItem;
  mHead:= mHead^.mNext;
  Dispose(r);
  Dec(mCount);
  // Обновляем указатель на хвост
  if mCount=0 then mQue:= nil;
end;

// Извлечение из стека (синоним Get)

function TBuffer.Pop: TItem;
begin
  Result:= Get;
```

```
end;

// Доступ к очереди по индексу

function TBuffer.GetByIndex(aIndex: integer): TItem;
var  r: PStackRec;
begin
  Result:= nil;
  if (aIndex<1) or (aIndex>mCount) then Exit;
  r:= mHead;
  while Assigned(r) do begin
    Dec(aIndex);
    if aIndex=0 then Break;
    r:= r^.mNext;
  end;
  if Assigned(r) then Result:= r^.mItem;
end;

// Реверс буфера (перестановка в обратном порядке)

procedure TBuffer.Reversion;
var B : TBuffer;
    t : TItem;
begin
  B:= TBuffer.Create;
  t:= Get;
  while Assigned(t) do begin   B.Push(t);  t:= Get;  end;
  t:= B.Get;
  while Assigned(t) do begin   Put(t);   t:= B.Get;  end;
  B.Free;
end;

// Печать буфера

procedure TBuffer.Print(var aFile: TextFile);
var r: PStackRec;
begin
  Writeln(aFile, '(');
  r:= mHead;
  while Assigned(r) do begin
    r^.mItem.Print(aFile);
    r:= r^.mNext;
  end;
  Writeln(aFile, ') BufCount= ', mCount);
end;
```

5.4. Демонстрация

Следующая программа демонстрирует возможности универсального буфера.

Листинг 5-2 — Программа для демонтрации возможностей буфера

```
{$APPTYPE CONSOLE}
uses
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas';

var Buf : TBuffer;
      T : TItem;
begin
  Buf:= TBuffer.Create;  // создание буфера

  // Создание и вставка элементов в конец буфера:
  // символы
  Buf.Put(TItemChar.Create('A'));
  Buf.Put(TItemChar.Create('B'));
  Buf.Put(TItemChar.Create('C'));
  // числа
  Buf.Put(TItemNum.Create(10));
  Buf.Put(TItemNum.Create(20));
  Buf.Put(TItemNum.Create(30));
  // Вывод на экран:
  Buf.Expo; Writeln;

  // Доступ по индексу:

  T:= Buf.GetByIndex(1);  // 1-й элемент
  T.Expo;
  T:= Buf.GetByIndex(4);  // 4-й элемент
  T.Expo;
  Writeln;

  // Реверс:
  Buf.Reversion;        // 1-й реверс
  Buf.Expo; Writeln;    // вывод
  Buf.Reversion;        // 2-й реверс
  Buf.Expo; Writeln;    // вывод

  // Круговой сдвиг на три позиции:
  // (выбор с начала и вставка в конец)
  Buf.Put(Buf.Get);
  Buf.Put(Buf.Get);
  Buf.Put(Buf.Get);
  Buf.Expo; Writeln;    // вывод

  // Извлечение из стека, вывод и уничтожение объектов

  while Buf.GetCount<>0 do begin
    T:= Buf.Pop;
    T.Expo;
    T.Free;
  end;
  Writeln;

  // Очистка буфера с уничтожением объектов:
  Buf.ClrAndDestroy;
  Buf.Free;
  Readln;
end.
```

5.5. Итоги

5.5.1. Универсальный буфер — это *элемент*, предназначенный для хранения других элементов (в том числе других буферов).

5.5.2. Универсальный буфер обладает возможностями безразмерного динамического массива, очереди и стека.

Глава 6
Представление множеств

Теперь приступаем к постройке *объекта-множества.* Множество — инструмент классификации наших представлений об окружающем мире — лежит в основе современной математики. Объекты-множества потребуются нам при решении последующих задач на *графах*.

6.1. Множества в Паскале

Тип *множество* встроен в язык Паскаль, где предусмотрены базовые операции с ним: объединение, вычитание, пересечение, сравнение. Но возможности встроенных в Паскаль множеств весьма ограничены, а именно:

- элементами множеств в Паскале могут быть только простые типы данных: символы и числа;
- максимальное количество элементов (мощность множества) невелико и обычно ограничено 256 элементами.

Для решения широкого круга задач нужен такой тип множества (класс), где эти оковы исчезнут. Методы создаваемого нами класса обеспечат исполнение стандартных операций с множествами, а также ряд вспомогательных действий. Ёмкость множества будет ограничена лишь объёмом памяти, а элементами множеств будут любые объекты, включая сами множества.

6.2. Объявление абстрактного множества TSet

Требования к создаваемому классу видны из его объявления, где перечисляют все желаемые операции над множествами. Класс *множество* будет произведён нами от класса *элемент*, а это значит, что сами множества можно трактовать и как элементарные объекты. Вот объявление класса:

```
// Класс "абстрактное множество"

  TSet = class (TItem)
    protected
      mCount: Longint;  // количество элементов множества
      procedure Clr(aDestroy: boolean); virtual; abstract;
    public
      // Базовые операции с множествами:
      procedure Add(arg : TSet); virtual; abstract;  // объединение
      procedure Sub(arg : TSet); virtual; abstract;  // вычитание
      procedure Mul(arg : TSet); virtual; abstract;  // пересечение
      procedure ExOr(arg : TSet); virtual; abstract; // исключающее ИЛИ
      function  TestIntersect(arg: TSet): boolean;   // проверка пересечения
      function  CompareSet(arg: TSet): TCompare;
      function  Compare(arg: TItem): TCompare; override;
      function  Exist(arg : TItem): boolean; virtual; abstract;
      // Очистка
      procedure Clear;          // очистка без уничтожения элементов
      procedure ClrAndDestroy;  // очистка с уничтожением элементов
```

```
    // Вставка, удаление, копирование элементоа
    function  Insert(arg: TItem): boolean; virtual; abstract;
    procedure Delete(arg: TItem); virtual; abstract;
    procedure CopyItems(arg: TSet);
    // Последовательный перебор элементов
    function  GetFirst: TItem; virtual; abstract; // выбрать первый
    function  GetNext: TItem; virtual; abstract;  // выбрать следующий
    // Выбор по индексу и по элементу
    function  GetItem(index: integer): TItem;
    function  GetObject(aItem: TItem): TItem;
    // Сохранение-восстановление позиции перебора
    procedure PositionPush;  virtual; abstract;
    procedure PositionPop;  virtual; abstract;
    // Прочие операции
    procedure CoverToDissect; // Преобразование покрытия в разбиение
    function  GetCount: integer;
    function  Copy: TItem; override;
    procedure Print(var aFile: TextFile); override;
    destructor Destroy;  override;
end;
```

Многие методы объявлены абстрактными, а это значит, что в этом базовом классе **`TSet`** они не будут реализованы, что не запрещает нам предварительно ознакомиться с ними.

6.2.1. Основные операции

Основные операции с множествами представлены в табл. 6-1, где для сравнения даны эквивалентные конструкции, записанные на Паскале.

Табл. 6-1 — Основные операции с множествами

Наименование	Эквивалент в Паскале	Вызов метода класса
Объединение	A:= A + B	A.Add (B)
Вычитание	A:= A - B	A.Sub (B)
Пересечение	A:= A * B	A.Mul (B)
Исключающее ИЛИ	A:= A + B – A * B	A.ExOr (B)
Проверка пересечения	A * B <> [ ]	A.TestIntersect (B)
Сравнение	A = B A < B A > B	A.CompareSet (B)
Проверка элемента	x in A	A.Exist (x)
Вставка элемента	A:= A + [x]	A.Insert (x)
Удаление элемента	A:= A – [x]	A.Delete (x)
Копирование всех элементов множества	A:= B	A.CopyItems (B)
Очистка множества	A:= [ ]	A.Clear A.ClrAndDestroy

Очищать множества можно двояко: либо с уничтожением, либо без уничтожения элементов множества. Деструктор множества предварительно очищает его без уничтожения включенных в него элементов.

6.2.2. Перебор элементов

В классическом множестве не важен порядок его элементов. Однако на практике такой порядок обычно существует, как и потребность в переборе элементов, и в доступе к отдельным элементам множества по индексу (как в массиве). Представленные ниже методы класса дают эти возможности.

Табл. 6-2 — Методы доступа к элементам множества TSet

Метод класса	Описание
A.GetCount	Возвращает количество элементов множества.
A.GetItem(i)	Возвращает i-й элемент множества, где i = 1...GetCount. Если индекс выходит за допустимые пределы, возвращается пустое значение NIL.
A.GetFirst	Устанавливает позицию внутреннего указателя на первый элемент множества и возвращает первый элемент, либо NIL для пустого множества.
A.GetNext	Продвигает позицию внутреннего указателя на следующий элемент множества и возвращает следующий элемент, либо NIL, если следующий элемент не существует.
A.PositionPush	Сохраняет текущее значение внутренней позиции чтения.
A.PositionPop	Восстанавливает предыдущее значение внутренней позиции чтения.
A.GetItem(x)	Возвращает указатель на элемент множества, совпадающий с объектом x.

Первыми двумя методами можно перебирать элементы множества по схеме, характерной для массива:

```
for i:=1 to A.GetCount do begin
  x:= A.GetItem(i);
  { обработать элемент x }
end;
```

На первый взгляд кажется, что этого достаточно на все случаи жизни, но мы не касались ещё реализации множеств... Слегка забегая вперёд, отмечу, что в нашей реализации выбор i–го элемента влечёт перебор по цепочке предыдущих i-1 элементов, а это накладно. Проблему решают два метода, применяемые по такой схеме:

```
x:= A.GetFirst;               // взять первый элемент
while Assigned(x) do begin
  { обработать элемент x }
  x:= A.GetNext;              // взять следующий элемент
end;
```

Методы **`GetFirst`** и **`GetNext`** используют внутренний указатель на текущий элемент. Но тут возникает другая проблема: как быть с вложенными переборами? Её решают методы сохранения и восстановления текущей позиции чтения:

```
X:= A.GetFirst;     // взять первый
while Assigned(x) do begin
  A.PositionPush;  // сохранить позицию чтения
  { обработать элемент X, при этом позиция чтения может нарушиться }
  A.PositionPop;    // восстановить позицию чтения
  x:= A.GetNext;    // взять следующий
end;
```

Или в других ситуациях так:

```
A.PositionPush;     // сохранить позицию чтения
X:= A.GetFirst;     // взять первый
while Assigned(x) do begin
  { обработать элемент X }
  X:= A.GetNext;    // взять следующий
end;
A.PositionPop;      // восстановить позицию чтения
```

Наконец метод **GetItem(X)** выбирает из множества элемент, эквивалентный элементу **X** (если сравнивать их методом **Compare**). Если элемент **X** уже является элементом массива, то возвращается сам элемент **X.** Если эквивалентного элемента в множестве нет, возвращается **NIL**.

6.2.3. Прочие методы

Прочие методы реализуют методы, унаследованные от базового класса *элемент* (такие, как **Copy**, **Print**, **Destroy**). Метод **CoverToDissect** будет рассмотрен позже.

6.3. Реализация класса TSet

В объявлении класса **TSet** многие методы объявлены абстрактными. Пока их нельзя реализовать, поскольку мы не определили структуру данных, на которой будут построены множества. Можно предложить несколько таких структур, каждая из которых обладает своими преимуществами и недостатками, как то: динамический массив, дерево, список. Лучшую из них в конкретных условиях может выявить только эксперимент. Хотя объекты абстрактного класса **TSet** никогда не будут созданы и применены, класс этот станет родителем реальных классов-множеств. Эта абстракция освобождает руки тем, кто захочет реализовать множества как-то иначе, а не так, как сделано в этой книге. И все эти возможные реализации будут пригодны для решения последующих задач.

И всё же, не смотря на абстрактность класса **TSet**, часть его методов будут реализованы уже сейчас. Этим мы услужим создателям классов-потомков (в том числе и себе), поскольку чем больше методов реализовано в предке, тем меньше возни с потомками.

Методы, реализованные в классе **TSet**, объявлены статическими, ознакомиться с ними можно по листингу модуля **Root**. В качестве примера ниже дана реализация метода сравнения множеств:

```
// Сравнение двух множеств
// Возвращаемые значения:
// cmpEq      - множества совпадают
// cmpLess    - первое является подмножеством второго
// cmpGreate  - второе является подмножеством первого
// cmpIncomp  - множества не поглощают друг друга

function TSet.Compare(arg: TItem): TCompare;
var p, q : TItem;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // один и тот же объект

  // Если аргумент не множество, то сравнивать нельзя
  if not (arg is TSet) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;

  // Сравнение элементов множеств
  PositionPush;             // сохр. позицию в данном множестве
  (arg as TSet).PositionPush;  // сохр. позицию в аргументе
  p:= GetFirst;             // взять первый элемент в данном множестве
  q:= (arg as TSet).GetFirst;  // взять первый элемент в аргументе
  while Assigned(p) and Assigned(q) do begin  // пока можно сравнивать
    Result:= p.Compare(q);                    // сравнение элементов
    if Result <> cmpEq then Break;            // выход, если не одинаковы
    p:= GetNext;  q:=(arg as TSet).GetNext;   // следующая пара
  end;
  if Result=cmpEq then begin                // если последние эл. совпали
    if GetCount < (arg as TSet).GetCount
      then Result:= cmpLess                 // данный меньше аргумента
    else if GetCount > TSet(arg).GetCount
      then Result:= cmpGreate               // данный больше аргумента
  end;
  (arg as TSet).PositionPop;    // восст. позицию в аргументе
  PositionPop;              // восст. позицию в данном множестве
end;
```

6.4. Класс TSetList

Напомню, что постройка абстрактного класса **`TSet`** преследовала две цели:

- объявить все методы, свойственные множествам, отвязав их от реализации;
- реализовать некоторые методы в базовом классе для облегчения последующей разработки потомков.

Теперь можно сконструировать потомка (одного из возможных), в котором будут реализованы все методы множества. Для хранения элементов множества воспользуемся относительно простой структурой — ***ОДНОСВЯЗНЫМ СПИСКОМ***. Возможно, что это решение не всегда будет оптимальным, но в большинстве случаев оно вполне приемлемо.

В объявление класса-потомка поместим нужные для списка поля, а все абстрактные методы класса-предка переопределим:

```
// Структура для организации односвязного списка

  PListRec = ^TListRec;
  TListRec = record
    mItem : TItem;
    mNext : PListRec;
  end;

// Множество, реализованное списком

  TSetList = class (TSet)
    protected
      mHead: PListRec;   // голова списка
      procedure Clr(arg: boolean); override;
    private
      mCurrent : PListRec; // указатель на текущий элемент
      mStack : TBuffer;  // стек для хранения позиций чтения mCurrent
      procedure Del(var p, q : PListRec);
      procedure Ins(p: PListRec; var q: PListRec; arg: TItem);
    public
      // переопределённые методы предка
      destructor Destroy;  override;
      function Exist(arg : TItem): boolean; override;
      function Insert(arg: TItem): boolean; override;
      procedure Delete(arg: TItem); override;
      function GetFirst: TItem; override;
      function GetNext: TItem; override;
      procedure PositionPush;  override;
      procedure PositionPop;   override;
      procedure Add(arg : TSet); override;
      procedure Sub(arg : TSet); override;
      procedure Mul(arg : TSet); override;
      procedure ExOr(arg : TSet); override;
  end;
```

Надо отметить, что многие переопределённые здесь методы можно было бы тоже объявить статическими и реализовать в родительском классе. Но такое решение не учло бы тонкостей работы с односвязным списком и повлекло бы заметное снижение быстродействия.

Для ознакомления с переопределёнными методами отсылаю вас к листингу модуля **SetList**, а в качестве примера привожу метод объединения множеств:

```
procedure TSetList.Add(arg: TSet);
var p, q : PListRec;
    t : TItem;
    compare : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not (arg is TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    compare:= cmpEq;
    while Assigned(p) do begin
      compare:= p^.mItem.Compare(t);
      case compare of
        cmpLess  : begin q:= p; p:= p^.mNext end;
```

```
        cmpEq    : begin t:= arg.GetNext; break end;
        cmpGreate,
        cmpIncomp: Break;
      end;
    end;
    if not Assigned(p) or (compare = cmpGreate) then begin
      Ins(p,q,t);
      if Assigned(t) then t:= arg.GetNext;
    end;
  end;
  arg.PositionPop;
end;
```

В следующей главе мы испытаем классы **TSet** и **TSetList**.

6.5. Итоги

6.5.1. Встроенный в язык Паскаль тип *множество* ограничен типами хранимых элементов (символы и числа), и их количеством (до 256).

6.5.2. В абстрактном классе **TSet** объявлены все связанные с ним операции. Абстрактное множество может содержать любые объекты, а ёмкость его ограничена лишь объёмом доступной памяти.

6.5.3. В классе **TSet** не определён механизм хранения элементов, поэтому часть его методов объявлена абстрактными. Объекты класса **TSet** никогда не создаются.

6.5.4. В классе-потомке **TSetList** определён механизм хранения элементов — это односвязный список. В потомке переопределены и реализованы основные методы обработки множеств наиболее эффективным (для выбранной структуры) способом.

Глава 7
Операции с множествами

В дальнейшем мы будем очень плотно работать с множествами, и потому сначала опишем основные операции над ними, уделив внимание ряду важных технических деталей.

7.1. Множество символов

Начнём с примера (листинг 7-1), где элементами множеств будут объекты-символы (но не элементарные символы, как в Паскале). Описание класса **`TItemChar`** дано в модуле **`Items`**, а далее следуют подробные комментарии.

Листинг 7-1 — Действия с множеством символов

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Root in '..\Common\Root.pas',
  Items in '..\Common\Items.pas';

// Создание множества символов на основе строки

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
begin
  Result:= CreateSet;   // создание пустого множества
  for i:= 1 to Length(arg) do begin
    t:= TItemChar.Create(UpCase(arg[i]));  // создаём объект-символ
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

var A, B, C : TSet;    // множества объявлены абстрактными

begin
    // Создаём три множества:
    A:= MakeSet('Pascal');  // A = { A,C,L,P,S }
    B:= MakeSet('Delphi');  // B = { D,E,H,I,L,P }
    C:= CreateSet;          // C = { }
    Write('      A = '); A.Expo;
    Write('      B = '); B.Expo;
    // Объединение:
    Write('    A+B = ');
    C.CopyItems(A);
    C.Add(B);
    C.Expo;                { A,C,D,E,H,I,L,P,S }
    // Разность:
    Write('    A-B = ');
    C.CopyItems(A);
    C.Sub(B);
    C.Expo;                { A,C,S }
    // Пересечение:
    Write('    A*B = ');
    C.CopyItems(A);
```

```
    C.Mul(B);
    C.Expo;                 { L,P }
    // Исключающее ИЛИ:
    Write('A xor B = ');
    C.CopyItems(A);
    C.ExOr(B);
    C.Expo;                 { A,C,D,E,H,I,S }

    // Освобождение множеств:
    C.Free;             // здесь элементы не уничтожаем
    B.ClrAndDestroy;    // предварительно уничтожаем элементы
    B.Free;             // затем множество
    A.ClrAndDestroy;    // предварительно уничтожаем элементы
    A.Free;             // затем множество
    Readln;
end.
```

Итак, в начале программы видим функцию, создающую множество символов, выбираемых из параметра **arg**:

```
function MakeSet(const arg: string): TSet;
```

Обратите внимание, что типом результата является абстрактный класс **TSet**. *Важно понять,* что при объявлении функций, переменных и параметров мы всегда будем использовать именно абстрактный класс **TSet**. Это избавит наши программы от привязки к конкретной реализации класса-множества. Первый оператор в теле функции создаёт пустое множество, которое будет возвращено в качестве результата:

```
  Result:= CreateSet;   // создание пустого множества
```

Откуда взялась функция **CreateSet**? Она определена в модуле **Root** следующим образом:

```
function CreateSet: TSet;
begin
  Result:= TSetList.Create;
end;
```

Здесь кроется ещё одна маленькая хитрость: тело этой функции — единственное место, где вызывается конструктор потомка, реализующего класс-множество (в данном случае — **TSetList**). Если в будущем вы реализуете множество как-то иначе, вам придётся исправить в модуле **Root** только одну эту строку.

Вернёмся к функции **MakeSet**, где в цикле создаются элементы-символы **t** и вставляются в множество-результат. Здесь важен условный оператор:

```
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
```

Дело в том, что если при вставке в множество там окажется дубликат (копия вставляемого элемента), то вставка не случится, и тогда надо удалить ненужный объект-дубликат.

Следующее далее объявление трёх глобальных переменных

```
var A, B, C : TSet;     // множества объявлены абстрактными
```

по указанным выше причинам (для отвязки от реализации класса) опять содержит абстрактный класс **TSet**.

В теле главной программы показаны основные операции. Они выполняются с множеством **C**, куда предварительно копируются элементы из множества **A**. Обратите внимание, что метод **CopyItems** копирует указатели на объекты из другого множества, он не создаёт копий объектов. То же касается и метода **Add**, добавляющего элементы из другого множества.

Последний важный момент касается удаления объектов-множеств.

```
    C.Free;           // здесь элементы не уничтожаем
    B.ClrAndDestroy;  // предварительно уничтожаем элементы
    B.Free;           // затем множество
    A.ClrAndDestroy;  // предварительно уничтожаем элементы
    A.Free;           // затем множество
```

При уничтожении множества **C** нет нужды уничтожать содержащиеся в нём объекты, поскольку там хранятся лишь копии указателей на них. Что касается множеств **A** и **B**, то здесь методом **ClrAndDestroy** предварительно надо уничтожить сами объекты-символы, иначе они останутся в памяти, засоряя её вплоть до завершения программы.

7.2. Множество строк

Следующий пример демонстрирует те же операции, но выполняемые с множеством объектов-строк — слов, извлечённых из текстовой строки.

Листинг 7-2 — Действия с множеством слов (строк)

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas';

// Создание множества слов, извлечённых из строки:

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
    s: string;
begin
```

```
  Result:= CreateSet;   // создание пустого множества
  i:=1;
  while i<= Length(arg) do begin
    // ищем первый не пробел
    while (arg[i]=#32) and (i< Length(arg)) do Inc(i);
    s:='';
    // читаем слово до пробела
    while (arg[i]<>#32) and (i<= Length(arg)) do begin
      s:= s + Upcase(arg[i]);
      Inc(i);
    end;
    Inc(i);
    t:= TItemStr.Create(s);  // создаём объект-строку
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

var A, B, C : TSet;   // множества объявлены абстрактными

begin
    Assign(Output, 'Out.txt'); Rewrite(Output);
    A:= MakeSet('while if not begin var function string  end');
    B:= MakeSet('while if not integer const function string');
    C:= CreateSet;   // C = {}
    Write('      A = '); A.Expo;
    Write('      B = '); B.Expo;
    // Объединение:
    Write('    A+B = ');
    C.CopyItems(A);    C.Add(B);    C.Expo;
    // Разность:
    Write('    A-B = ');
    C.CopyItems(A);    C.Sub(B);    C.Expo;
    // Пересечение:
    Write('    A*B = ');
    C.CopyItems(A);    C.Mul(B);    C.Expo;
    // Исключающее ИЛИ:
    Write('A xor B = ');
    C.CopyItems(A);    C.ExOr(B);    C.Expo;

    // Освобождение множеств:
    C.Free;           // здесь элементы не уничтожаем
    B.ClrAndDestroy;  // предварительно уничтожаем элементы
    B.Free;
    A.ClrAndDestroy;  // предварительно уничтожаем элементы
    A.Free;
    Close(Output);
    Readln;
end.
```

Программа вывела в файл **Out.txt** следующий результат (после двоеточия даётся количество элементов множества):

```
      A = { BEGIN END FUNCTION IF NOT STRING VAR WHILE } : 8
      B = { CONST FUNCTION IF INTEGER NOT STRING WHILE } : 7
    A+B = { BEGIN CONST END FUNCTION IF INTEGER NOT STRING VAR WHILE } : 10
    A-B = { BEGIN END VAR } : 3
    A*B = { FUNCTION IF NOT STRING WHILE } : 5
A xor B = { BEGIN CONST END INTEGER VAR } : 5
```

7.3. Сравнение множеств

Следующий пример демонстрирует сравнение множеств методом **Compare** и проверку на пересечение методом **TestIntersect**.

Листинг 7-3 — Сравнение множеств

```
{$APPTYPE CONSOLE}

uses  Items in '..\Common\Items.pas',
      Root in '..\Common\Root.pas';

// Создание множества символов на основе строки

function MakeSet(const arg: string): TSet;
var i: integer;
    t: TItem;
begin
  Result:= CreateSet;   // создание пустого множества
  for i:= 1 to Length(arg) do begin
    t:= TItemChar.Create(UpCase(arg[i]));  // создаём объект-символ
    if not Result.Insert(t)  // если не удалось вставить в множество
      then t.Free;           // то удаляем дубликат
  end;
end;

// Вывод результата сравнения

procedure Show(arg: TCompare);
begin
  case arg of
    cmpEq:     Writeln('Eq');
    cmpLess:   Writeln('Less');
    cmpGreate: Writeln('Greate');
    cmpIncomp: Writeln('Incomp');
  end;
end;

var A, B, C, D : TSet;    // множества объявлены абстрактными

begin
    // Проверка на подмножество и надмножество
    A:= MakeSet('ABCD');
    B:= CreateSet;
    B.CopyItems(A);
    Show(A.Compare(B));  // Eq
    B:= MakeSet('BD');
    C:= MakeSet('DF');
    D:= MakeSet('FE');
    Show(A.Compare(B));  // Greate  (A - это надмножество B)
    Show(B.Compare(A));  // Less    (B - это подмножество A)
    Show(A.Compare(C));  // Incomp
    Show(A.Compare(D));  // Incomp
    // Проверка на пересечение
    Writeln(A.TestIntersect(B));  // TRUE - пересекаются
    Writeln(B.TestIntersect(A));  // TRUE - пересекаются
    Writeln(A.TestIntersect(D));  // FALSE - не пересекаются
    Readln;
end.
```

Метод сравнения **Compare** возвращает одно из четырёх значений:

cmpEq	множества совпадают
cmpLess	первое является подмножеством второго
cmpGreate	второе является подмножеством первого
cmpIncomp	множества не поглощают друг друга

Метод проверки на пересечение **TestIntersect** возвращает **TRUE**, если два множества пересекаются (содержат хотя бы один общий элемент).

7.4. Итоги

7.4.1. Объявления множеств — переменных, параметров и типов функций — выполняются через абстрактный класс **TSet**, что избавляет нас от привязки программ к конкретной реализации класса-множества.

7.4.2. Все объекты-множества будут создаваться функцией **CreateSet** из модуля **Root**. Тело этой функции — единственное место, где указан конкретный класс, реализующий множество (TSetList).

7.4.3. После вставки элемента методом **Insert** надо анализировать возвращаемый результат. Если метод вернул **FALSE**, значит, множество уже содержит такой элемент, и тогда можно удалить ненужный объект-дубликат.

7.4.4. При удалении множества методом **Free** автоматически вызывается метод **Clear**, очищающий множество, но не уничтожающий его элементы (элементы могут состоять в нескольких множествах сразу). Для уничтожения элементов множества вызывается метод **ClrAndDestroy**.

7.4.5. При сравнении множеств выясняется, совпадают ли они (cmpEq), либо одно является подмножеством другого (cmpLess, cmpGreate), либо множества не совпадают и не поглощают друг друга (cmpIncomp).

7.4.6. Методы класса **TSet** не привязаны к классам объектов, составляющих множество. Таким образом, операции с множествами действуют независимо от того, какие объекты содержатся в множествах.

Глава 8
Чудовище экспоненты

Отвлечёмся на время от множеств, чтобы обратиться к одной важной проблеме и двум методам её решения, — всё это нам скоро пригодится.

8.1. Непростая простота

Представим вполне вероятную картину. Вы проверили новую программу на небольшом наборе данных, скажем, на массиве из пяти элементов, а в следующем тесте «скармливаете» ей 10-ти элементный массив. Тут наблюдаете небольшое «торможение»: компьютер задумался на пару секунд. Ещё не чуя беды, вы предлагаете программе набор из двадцати элементов и... что это? Где результат? Вы ждёте его минуту, две, десять... Натужно гудит вентилятор процессора, а вы ждёте, и ждёте... Сколько ещё? Год или столетие? А может, компьютеру не хватит даже срока, отпущенного нашей Вселенной?! Программа зациклилась? Но прервав её и пройдя по шагам, вы убеждаетесь, что это не так. Наконец, выясняете, что нарвались на алгоритм, обладающий *экспоненциальной* сложностью, в данном случае — *по времени*. Другие программы такого рода могут завершаться относительно скоро, но аварийно — из-за нехватки памяти. Тогда говорят об *экспоненциальной* сложности *по памяти*. Рассмотрим в этой связи два элементарных примера.

Вот задача: вывести на экран числа от 1 до **N**. Ясно, что это легко выполнимо для весьма больших значений **N**, а время решения зависит от этого числа линейно. Теперь слегка изменим формулировку: выведем числа от 1 до 10^{N}. То есть, **N** теперь — показатель степени при 10. Легко догадаться, что уже при небольших значениях параметра **N** компьютеру потребуется заметное время. Когда же параметр составит десятки, не говоря о сотнях... нет, не стану продолжать: вы и так поняли, что это задача *экспоненциальной* сложности по времени.

Вторую задачу поставим так: сгенерировать и сохранить в памяти список чисел от 1 до 10^{N}. Ясно, что это тоже *экспоненциально* сложная задача: и по времени, и по памяти. Хотя вы наверняка дождётесь её завершения, но даже для умеренных значений параметра **N** завершение будет аварийным.

Итак, как ни могуч ваш компьютер, он не всемогущ: существуют задачи, принципиально на нём не решаемые, хотя алгоритмы их решения на взгляд человека весьма просты. Наше открытие побуждает разобраться в *классах сложности* алгоритмов.

8.2. Кто круче?

Понятие «сложность» применительно к алгоритмам можно трактовать двояко. Человеку сложность видится в нагромождении конструкций языка: циклов, ветвлений, структур данных. Компьютер смотрит на это иначе и определяет

сложность потребными ресурсами: количеством операций и объёмом необходимой памяти. Эта разница «взглядов» ясно видна из показанных выше примеров. Строго говоря, компьютер «напрягает» не сложность, а *трудоёмкость* алгоритма. Однако в литературе укоренился термин *сложность*, поэтому в дальнейшем мы будем говорить о *классах сложности* алгоритмов.

Ясно, что любой алгоритм обрабатывает десяток элементов быстрее, чем сотню: сложность задачи пропорциональна объёму обрабатываемых данных. На практике важно знать, насколько крута эта зависимость, именно эта *крутизна* делит алгоритмы на классы сложности. Ниже мы рассмотрим несколько таких классов в порядке возрастания их сложности.

8.2.1. Константная сложность

Пусть задача состоит в том, чтобы выбрать из массива размерности **N** элемент с индексом **i**. Известно, что время выполнения этой операции не зависит ни от размера массива **N**, ни от индекса **i**, то есть, является константой. Этот факт можно выразить формулой:

$$T(N) = Const$$

где T — это время выполнения алгоритма как функция от объёма данных **N**.

Алгоритм константной сложности — мечта программиста, однако большинство алгоритмов — увы! — куда сложнее!

8.2.2. Линейная сложность

Время исполнения таких алгоритмов зависит от объёма данных линейно, что можно выразить формулой:

$$T(N) = C \bullet N$$

где C — некоторая константа, определяемая характеристиками компьютера и другими факторами. Вот три примера алгоритмов этого класса: линейный поиск в массиве, выбор **i**–го элемента в односвязном списке, перебор элементов массива.

8.2.3. Квадратичная сложность

К этому классу относится, к примеру, алгоритм перебора элементов квадратных матриц, а также некоторые алгоритмы сортировки массива. Соответствующая им формула такова:

$$T(N) = C \bullet N^2$$

8.2.4. Сложности б**о**льших степеней

Квадратичная сложность — не предел. Некоторые алгоритмы обработки матриц обладают кубической сложностью и сложностью 4-й степени. Ещё реже встречаются сложности 5-й и 6-й степеней. Все их, включая линейную и квадратичную можно выразить формулой:

$$T(N) = C \bullet N^{k}$$

где $k = 1, 2, 3$ и т. д.

8.2.5. Логарифмическая и промежуточные сложности

Самый известный алгоритм логарифмической сложности — двоичный поиск в сортированном массиве; время поиска выражается формулой:

$$T(N) = C \bullet Log(N)$$

Алгоритмы этого класса весьма привлекательны, ведь логарифмическая функция растёт очень медленно. Это побуждает программистов искать пути совершенствования алгоритмов, сводя линейную сложность к логарифмической, или квадратичную — к промежуточной между линейной и квадратичной, что соответствует формуле:

$$T(N) = C \bullet N \bullet Log(N)$$

Этой формулой оценивают сложность алгоритма быстрой сортировки Хоара.

8.2.6. Полиномиальная сложность

Подведём промежуточный итог нашего обзора: на рис. 8-1 представлены графики нескольких из представленных выше функций. Видно, что функции б**о**льшей крутизны при возрастании **N** рано или поздно обгоняют менее «крутые». По этой причине при сравнении классов сложности пренебрегают постоянным множителем, а когда сложность алгоритма выражается суммой упомянутых выше формул, то учитывают лишь наиболее «крутую» составляющую. В итоге получаем следующую градацию классов в порядке возрастания сложности:

1. $\mathbf{Const}$
2. $\mathbf{C \bullet Log(N)}$
3. $\mathbf{C \bullet N}$
4. $\mathbf{C \bullet N \bullet Log(N)}$
5. $\mathbf{C \bullet N^2}$
6. $\mathbf{C \bullet N^2 \bullet Log(N)}$

и т.д.

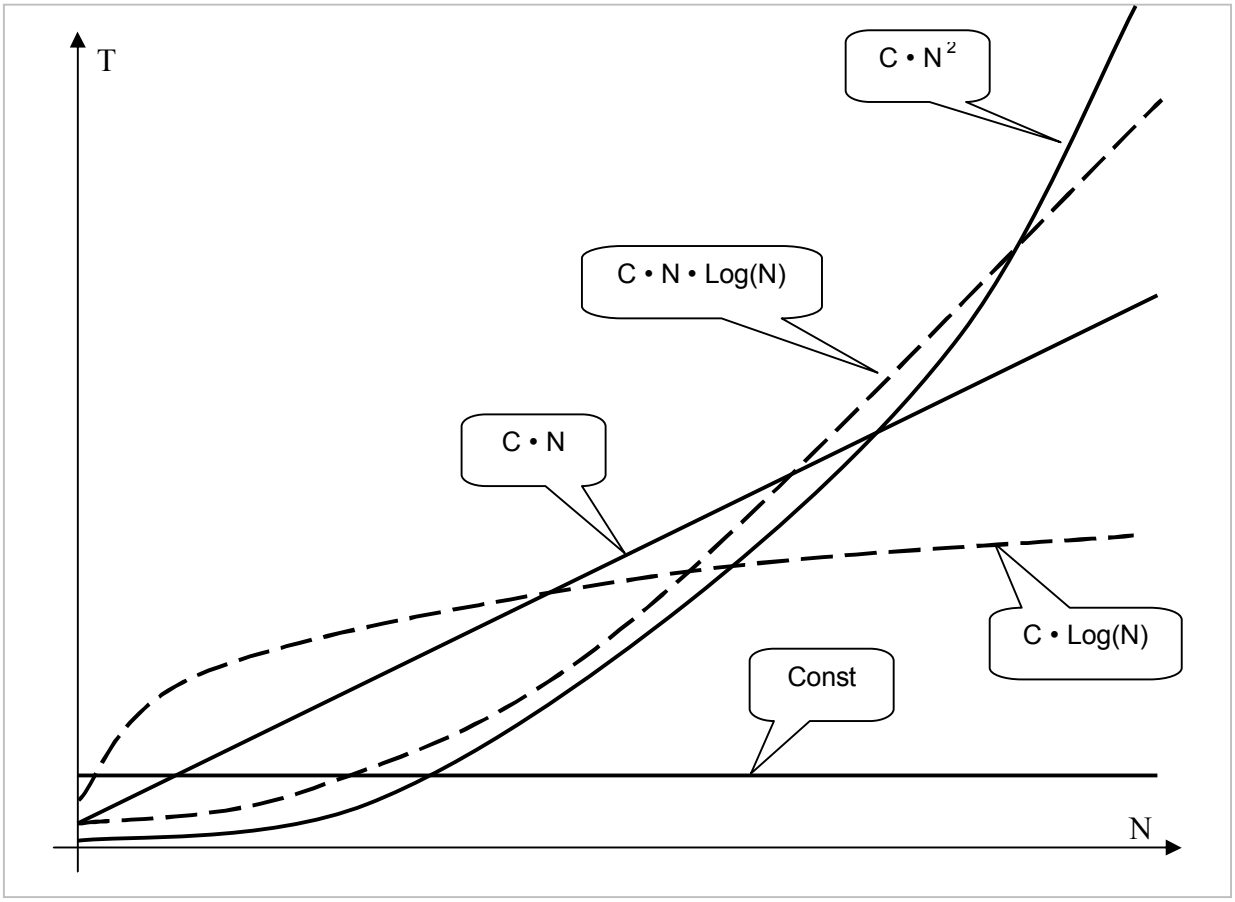


Рис. 8-1 — Графики зависимости трудоёмкости от объёма данных N для нескольких классов алгоритмов

Ясно, что программист всегда ищет наименее сложный алгоритм. Но сейчас важно понять другое: перечисленные выше классы составляют категорию реально выполнимых *полиномиальных* алгоритмов. Это значит, что такие задачи могут быть решены на компьютере за разумное время для всех практически необходимых наборов данных. По крайней мере, не сегодня, так завтра. Теперь обратимся к более «крутым» классам задач.

8.2.7. Экспоненциальная сложность

К этой категории относят алгоритмы, время исполнения которых (или объём необходимой им памяти) растут как показательная функция, или ещё круче: факториал и т.д., например:

$T(N) = C^N$ — *показательная функция*

$T(N) = C \cdot N!$ — *факториал*

$T(N) = C \cdot N^N$

Известно, что как бы круто ни рос полином, показательная функция с возрастанием **N** рано или поздно превзойдёт его при любых постоянных

коэффициентах. Две прочие функции ещё «круче». Но нас, программистов, это не радует. Эта «крутизна» означает, что решение многих таких задач на практике невозможно. Или почти невозможно. Далее мы рассмотрим две идеи, дающие надежду на преодоление этой проблемы: *поиск с возвратом* и *жадные* алгоритмы.

8.3. Поиск с возвратом

Алгоритмы экспоненциальной сложности часто сопряжены с обработкой древовидных структур, где на каждом уровне дерева возможен выбор одного из **N** вариантов, а количество уровней составляет **M**. Время поиска решения (перебор всех вариантов) здесь будет пропорционально N^M. В других задачах такого рода величины **N** и **M** могут быть связаны соотношением **M=C•N**, и тогда время поиска решения будет пропорционально N^N. Рассмотрим несложный пример такой задачи.

Пусть рис. 8-1 изображает схему некой горнолыжной трассы. Воображаемый спортсмен стартует из верхней точки, обозначенной пустым кружком, и стремится к подножью горы, пересекая уровни **A**, **B** и **C**. Время движения между соседними промежуточными точками указано внутри кружков. Цель в том, чтобы найти быстрейший маршрут.

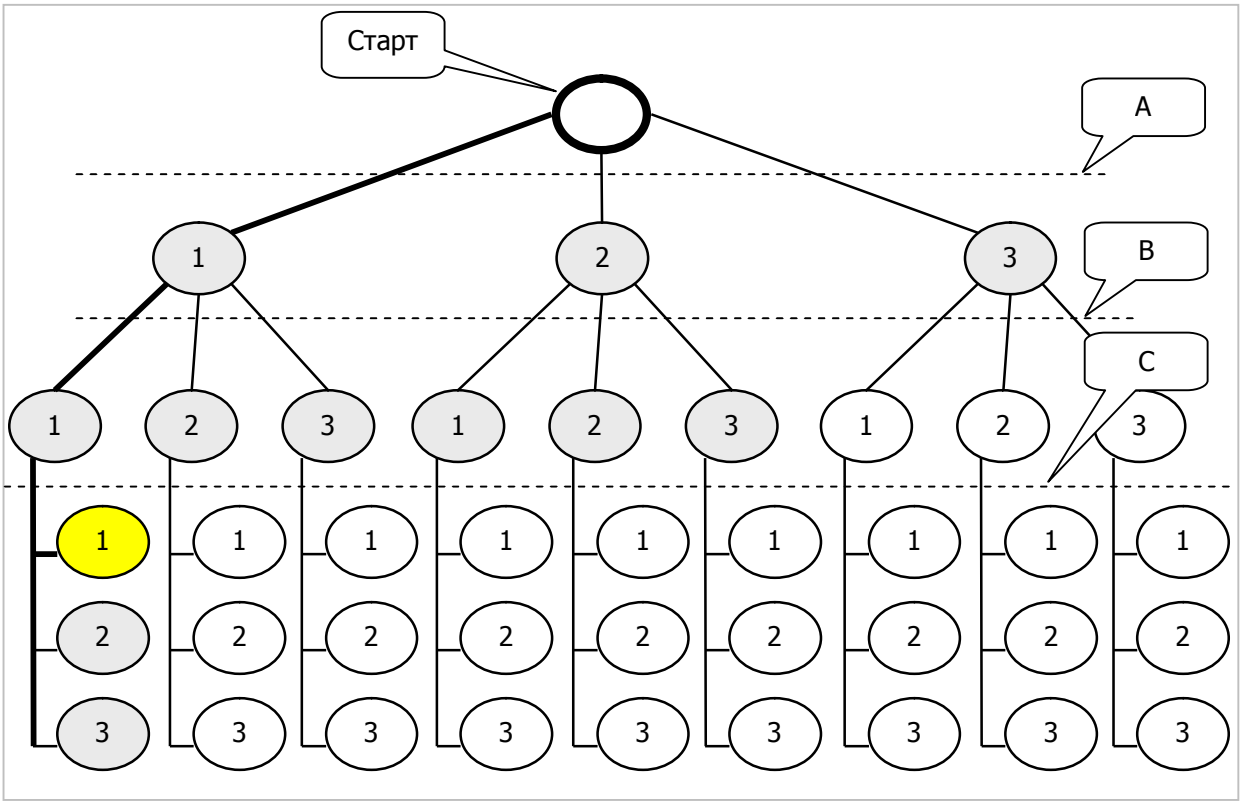


Рисунок 8-1 — Дерево поиска минимальной суммы ("счастливый" случай)

Здесь «спортсмен» будет перебирать все возможные варианты, количество которых зависит от ветвистости дерева и числа уровней, — это задача

экспоненциальной сложности. Однако не зря на рисунке времена расставлены в возрастающем порядке, — это поможет нам разобраться в *поиске с возвратом*.

Пусть «горнолыжник» на каждом уровне перебирает варианты слева направо. Тогда левую точку уровня **A** он достигнет за 1 минуту. Спустившись к уровню **B**, он достигнет крайнеё левой точки ещё через 1 минуту. Отсюда он попадает на уровень **C**, и здесь — о, удача! — первая точка снова оказывается ближайшей. В итоге первый спуск занял три минуты, запомним этот результат (он отмечен жёлтым). Однако, не будучи уверенным в его оптимальности, спортсмен должен опробовать ещё два маршрута, которые дадут времена соответственно **1+1+2** и **1+1+3** минуты (отмечены серым).

Затем «спортсмен» исследует другие маршруты. Двигаясь от точки **A1** к точке **B2** (нумеруем точки слева направо), он затратит **1+2** минуты, — а этот результат, заметим, уже равен оптимальному. Стало быть, спускаться отсюда на уровень **C** нет смысла. Это же решение он примет во всех последующих точках, где текущее время окажется равным, либо большим текущего лучшего результата. В итоге, вместо **3+9+27=39** точек лыжник исследует лишь **3+6+3=12** точек (посещённые точки отмечены серым).

Хорошо, но здесь трёхкратный выигрыш (**12** против **29**) был достигнут при удачном стечении обстоятельств, а если времена расположить в обратном порядке? Тогда лучший маршрут найдётся в последней попытке, и никакого выигрыша не будет. В реальных задачах и лучший, и худший исходы маловероятны. Зато при больших наборах случайных данных выигрыш может исчисляться многими порядками — веское основание, чтобы вооружиться методом *поиска с возвратом*.

8.4. Принцип жадности

Итак, хотя поиск с возвратом многократно ускоряет перебор вариантов, он не гарантирует двух вещей. Во-первых, не исключает худшего, когда приходится перебрать все возможные варианты. Во-вторых, что ещё важнее, в иных приложениях затраты времени на поиск идеала будут неприемлемы. Допустим, что подобная задача решается системой управления воздушной обороной, которая выбирает одну из нескольких целей — наиболее опасную. При этом учитывается много факторов, что делает задачу экспоненциально сложной. Стремясь к точному решению, система упускает драгоценное время, что лишает смысла весь расчёт. Не поискать ли решения, пусть и не идеального, но быстрого? Уж лучше сбить одну из опасных воздушных целей, чем пропустить их все.

Вернёмся к горнолыжному примеру, немного изменив данные на уровне **B**, как это показано на рис. 8-2. Теперь идеальный спуск следует по цепочке **A3** → **B1** → **C1**, а соответствующее время составит **3+1+1=5** минут. Однако найти его будет непросто, — придется перебрать море вариантов. Так испробуем путь, на который толкает, вероятно, сама наша природа: пусть лыжник сделает лишь одну попытку,

но на каждом уровне пробует не все варианты, а выбирает только тот, что сулит на данном этапе (уровне) лучшее время.

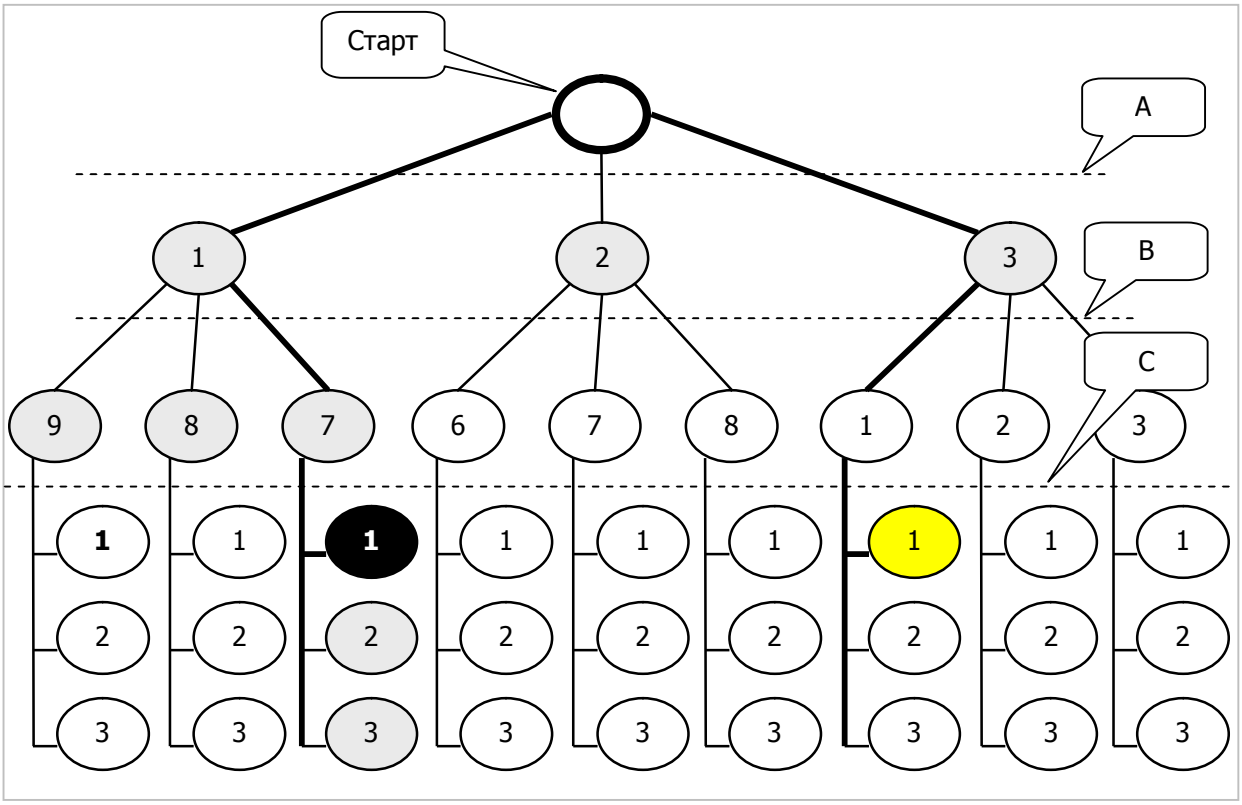


Рис. 8-2 — Дерево поиска с изменённым уровнем B

Следуя этому принципу *жадности*, на первом уровне он выберет точку **A1** (1 минута). Отсюда ему уже не попасть на оптимальную трассу, но жадность диктует своё: дальше он выберет точки **B7** (7 минут) и **C1** (1 минута), показав результат **1+7+1=9** минут, — не самый быстрый, но далеко и не худший. Зато как быстро решена задача! Всё потому, что на каждом уровне он обрабатывает только **N** узлов, что при **M** уровнях даёт величину порядка **N•M**, то есть время поиска здесь зависит линейно от ветвистости и глубины дерева. К тому же оно стабильно, то есть, не подвластно случайностям распределения данных по дереву.

А что на практике? Насколько результат жадного поиска хуже оптимального? Забегая вперёд, замечу, что в будущих опытах на случайных наборах данных мы получим результаты, уступающие оптимальным на 20-30%, что не так уж плохо. Иногда жадный поиск даёт идеальный результат, и никогда не ведёт к худшему. Окончательную оценку жадного принципа дают лишь эксперименты на конкретных данных.

8.5. Комбинации

Итак, мы рассмотрели два принципа борьбы с экспоненциальной сложностью: *поиск с возвратом* и *жадность*, каждый из которых обладает достоинствами и недостатками. Можно ли совместить их?

Возьмём ту же систему воздушной обороны. Пусть предел времени, отведенного на обработку данных и выбор подходящей цели, ограничен значением **T**. Тогда можно сначала применить жадный алгоритм и найти пусть не идеальное, но приемлемое решение. Затем дать волю поиску с возвратом, позволив ему работать в пределах времени **T**. Здесь в качестве первого приближения можно воспользоваться предварительным результатом жадного поиска. Если за время **T** перебор не завершится, то взять текущий лучший результат перебора.

8.6. Итоги

8.6.1. Время обработки данных пропорционально их объёму, но крутизна этой зависимости определяется алгоритмом. В этой связи все алгоритмы делятся на *классы сложности*.

8.6.2. Предпочтительно, если это возможно, сводить решения к алгоритмам низших классов: логарифмическим, линейным и т.д.

8.6.3. Классы сложности, где зависимость времени решения от объёма данных определяется полиномом и (или) логарифмом, называют *p-классами*. Соответствующие им задачи решаются на компьютере за разумное время с разумным расходом памяти.

8.6.4. Классы сложности, где время решения растёт от объёма данных по экспоненте или быстрее, называют *экспоненциальными*. Задачи такого рода, за недостатком ресурсов, обычно не могут быть решены «в лоб».

8.6.5. Для преодоления экспоненциального барьера используют два принципа: *поиск с возвратом* и *жадность*.

8.6.6. *Поиск с возвратом* состоит в отбрасывании ветвей, заведомо не ведущих к лучшему результату. На практике метод сокращает время решения на много порядков (в среднем).

8.6.7. Принцип *жадности* состоит в выборе на каждом этапе только одного из возможных вариантов, кажущегося лучшим на данном этапе. Это порождает очень быстрые алгоритмы, не гарантирующие идеального результата.

8.6.8. Жадные алгоритмы иногда дают идеальный результат, и никогда не дают худшего.

8.6.9. В некоторых ситуациях комбинируют *жадный* метод и *поиск с возвратом*, ограничивая его по времени.

8.7. Что почитать

№	Автор(ы)	Название	Главы, страницы
√ 1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
√ 6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 9
Разбиение множеств

9.1. Зачем разбивают множества?

Зачем множества разбивают на подмножества? — познания ради. Этим всегда безотчетно занимались животные: в стремлении выжить, они учились разбивать множества понятий на полезные подмножества, и от успеха этой «учёбы» зависела их судьба. Так, к примеру, глаз лягушки разделяет всё движущееся на два подмножества: мелкие предметы — пища, а крупные — опасность. Лягушке этого довольно. Высшие животные оказывают большую разборчивость и способны различать тысячи разных объектов, неосознанно формируя из них подмножества. Тем же, но сознательно, заняты учёные. К примеру, растения и животных они классифицируют по ряду признаков, разбивая их на подмножества. Вероятно, по той же причине ребёнок разбивает игрушку.

Хорошо, а зачем это нам? Во-первых, будучи истинными учёными, мы попытаемся сформировать все возможные разбиения некоторого множества — дело чести, однако. Во-вторых, разбиения, как исходный материал, пригодятся нам в последующих задачах, — для этого мы создадим процедуры для формирования случайных разбиений с заданными характеристиками.

9.2. Что такое разбиение?

Это поймёт любой, кто хоть однажды бил посуду. Разбиение — это то, из чего собирается целое, причём так, чтобы не оставалось лишних «деталей». Сколькими способами можно разбить целое? Кувшин бьётся бесчисленными способами. Но мы будем «бить» дискретные объекты — множества, и здесь количество осколков и их комбинаций, хоть и велико, но ограничено.

Рассмотрим пример разбиения множества из трёх символов: { a, b, c }. Сколькими способами оно бьётся? Во-первых, поскольку множество является собственным подмножеством, разбиение может состоять из самого множества и пустого подмножества, которое обычно игнорируют. Во-вторых, его можно разбить на подмножества по одному символу в каждом. Это два крайних варианта разбиений, но есть ещё три промежуточных, и все они показаны в табл. 9-1.

Табл. 9-1 — Варианты разбиения множества { a, b, c }

Разбиение	Количество подмножеств
{ {a, b, c} }	1
{ {a}, {b, c} }	2
{ {a, b}, {c} }	2
{ {a, c}, {b} }	2
{ {a}, {b}, {c} }	3

Итак, мы видим, что *разбиение* — это *множество подмножеств*, сумма которых даёт целое множество, причём все подмножества одного разбиения взаимно не пересекаются. Это видно из табл. 9-1 (учтите, что пустое подмножество неявно присутствует в первом разбиении).

Для облегчения восприятия далее я приму сокращённую запись множества множеств; в этой краткой форме удалены внутренние запятые и скобки, что показано в следующей сравнительной таблице.

Табл. 9-2 — Развёрнутая и краткая формы записи множества множеств

Развёрнутая форма	Краткая форма
{ {a, b, c} }	{ abc }
{ {a}, {b, c} }	{ a, bc }
{ {a, b}, {c} }	{ ab, c }
{ {a, c}, {b} }	{ ac, b }
{ {a}, {b}, {c} }	{ a, b, c }

9.3. Все разбиения

Здесь мы поставим и *почти* решим следующую задачу: составить для данного множества *все* возможные его разбиения. Каждое разбиение будет представлять собой множество подмножеств. Попутно выясним возможное количество разных разбиений, и тогда станет понятно, почему задача решается лишь «почти».

Идея алгоритма основана на последовательном расширении исходного множества (а начнём с пустого). Пусть на каком-то этапе известны все разбиения множества из предыдущих **`N-1`** элементов. Тогда все разбиения для множества из **`N`** элементов можно получить отсюда двумя способами:

1. *Вставить* очередной элемент в одно из подмножеств разбиения, — так последовательной вставкой элементов получим *несколько* разбиений.
2. *Добавить* к существующим подмножествам ещё одно подмножество, включающее лишь один элемент, — так получим ещё *одно* разбиение.

В табл. 9-3 дана последовательность построения всех разбиений для множества из четырёх элементов. Попутно строятся разбиения для множеств из одного, двух и трёх элементов (в таблице используется краткая форма записи множества подмножеств).

Табл. 9-3 — Построение всех разбиений последовательным расширением исходного множества

Исходные разбиения	Добавляемый элемент	Способы формирования и результаты разбиений		Кол-во разбиений
		1) Вставкой	2) Добавлением	
1-й элемент (a)				
{ }	a	нет	{ a }	1
2-й элемент (b)				
{ a }	b	{ ab }	{ a, b }	2
3-й элемент (c)				
{ ab }	c	{ abc }	{ ab, c }	5
{ a, b }		{ ac, b }	{ a, b, c }	
		{ a, bc }		
4-й элемент (d)				
{ abc }	d	{ abcd }	{ abc, d }	15
{ ac, b }		{ acd, b }	{ ac, b, d }	
		{ ac, bd }		
{ a, bc }		{ ad, bc }	{ a, bc, d }	
		{ a, bcd }		
{ ab, c }		{ abd, c }	{ ab, c, d }	
		{ ab, cd }		
{ a, b, c }		{ ad, b, c }	{ a, b, c, d }	
		{ a, bd, c }		
		{ a, b, cd }		

Итак, начнём с пустого множества, — его разбиение тоже пусто. Поэтому при обработке первого элемента нельзя создать разбиения вставкой (некуда вставлять). Потому ограничимся вторым способом: добавим множество из единственного элемента {a}.

При обработке второго элемента применимы оба способа, поэтому здесь получаем два разбиения: { ab } и { a, b }, — здесь и далее я использую краткую форму записи подмножеств. На следующем шаге эти два разбиения будут «сырьём» для построения разбиений множества из трёх элементов и т.д.

В правом столбце табл. 9-3 показано количество получаемых разбиений. Оно соответствует *числам Белла*, которые вычисляются как сумма *чисел Стирлинга* второго рода (подробнее об этом читайте в рекомендуемой литературе). В табл. 9-4 дана зависимость чисел Белла от количества элементов в разбиваемом множестве (его мощности). Здесь видно, что количество возможных разбиений растёт с ростом множества чрезвычайно быстро (быстрее экспоненты). Стало быть, задача формирования *всех* разбиений множества *экспоненциально сложна*, и может быть решена на практике лишь для небольших множеств.

Табл. 9-4 — Числа Белла B(n)

Мощность множества N	Количество возможных разбиений B(N)
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4 140
9	21 147
10	115 975
11	678 570
12	4 213 597
13	27 644 437
14	190 899 322
15	1 382 958 545
16	10 480 142 147
17	82 864 869 804
18	682 076 806 159
19	5 832 742 205 057
20	51 724 158 235 372

Но сложность задачи, даже экспоненциальная, не помешает нам создать решающую её процедуру. Точнее, это будет функция, принимающая аргумент-множество и возвращающая другое множество, состоящее из *всех* разбиений исходного. Отметим, что результат будет «множеством в кубе», поскольку его элементами будут разбиения, то есть множества, а те, в свою очередь, будут множествами подмножеств. Так, к примеру, для исходного множества {a,b} будет сформирован результат { { {a,b} }, { {a}, {b} } } —множество множеств, содержащих подмножества (дано в развёрнутой форме).

Итак, рассмотрим листинг функции **`GenAllDissections`** из модуля **`Dissect`**; некоторые моменты я прокомментирую подробней.

Листинг 9-1 — Формирование всех разбиений множества

```
function GenAllDissections(aSet: TSet): TSet;
var Que   : TBuffer;  // буфер-очередь
    SSGet : TSet;     // извлекаемое множество подмножеств
    SSPut : TSet;     // добавляемое множество подмножеств
    //-----------------------------------------
    // Процедура обработка всех подмножеств
    // из множества множеств SSGet

    procedure LocalHandle(aItem: TItem);
    var k: integer;
        S: TSet;     // текущее множество из множества подмножеств
    begin
      // Обработка очередного извлечённого множества подмножеств SSGet,
      // перебираем его элементы-множества S:
      for k:= 1 to SSGet.GetCount do begin
          SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
          S:= SSPut.GetItem(k) as TSet; // очередное множество из МП
          SSPut.Delete(S);   // удаляем из него очередное подмножество
          S.Insert(aItem);   // к очередному добавляем текущий элемент
          SSPut.Insert(S);   // полученное вставляем в множество подмножеств
          Que.Put(SSPut);    // новое множество подмножеств заносим в очередь
      end;
      // Добавление нового подмножества к SSGet
      SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
      S:= CreateSet;       // создаём множество
      S.Insert(aItem);     // из единственнного элемента aItem
      SSPut.Insert(S);     // добавляем к множеству подмножеств
      Que.Put(SSPut);      // новое множество подмножеств заносим в очередь
    end;
    //-----------------------------------------

var   i, j  : integer;  // счётчики
      Item: TItem;      // текущий элемент исходного множества

begin
  Que:= TBuffer.Create; // создание буфера для очереди
  SSPut:= CreateSet;       // пустое множество подмножеств
  Que.Put(SSPut);          // заносим в очередь
  // Цикл по всем элементам исходного множества:
  for i:= 1 to aSet.GetCount do begin
    Item:= aSet.GetItem(i);
    // Цикл обработки очереди:
    for j:=1 to Que.GetCount do begin
      SSGet:= Que.Get as TSet;  // берём очередное множество из очереди
      LocalHandle(Item);        // создаём из него ряд других
      SSGet.ClrAndDestroy;      // очищаем от подмножеств
      SSGet.Free;               // и уничтожаем
    end;
  end;
  Result:= CreateSet; // создание пустого результата
  // Перенос из буфера в множество-результат
  while Que.GetCount>0 do Result.Insert(Que.Get);
  Que.Free;  // удаление буфера
end;
```

В основном теле функции создаём рабочий буфер **Que**, — он будет служить очередью обрабатываемых разбиений. Вначале в очередь помещаем пустое множество **SSPut** — для «затравки».

Далее следует внешний цикл обработки элементов множества. Для каждого элемента организуем внутренний цикл, в котором обрабатываем все элементы очереди (все разбиения), находящиеся в ней на момент входа в этот внутренний цикл (при старте туда занесено одно пустое множество). Извлечённый из буфера элемент-разбиение **SSGet** обрабатываем локальной процедурой **LocalHandle**, а затем уничтожаем вместе с его содержимым.

Локальная процедура **LocalHandle** создаёт несколько новых разбиений из переданного ей разбиения по правилам, рассмотренным выше, и вставляет их в ту же очередь. Стало быть, очередь со временем разрастается по закону чисел Белла. После обработки всех элементов исходного множества результат переносим из очереди **Que** в множество **Result**, а очередь удаляем.

Теперь рассмотрим тестирующую программу. Внешне она проста, но некоторые моменты нуждаются в подробных разъяснениях (листинг 9-2).

Листинг 9-2 — Тестирующая программа для получения всех разбиений множества

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Dissect in '..\Common\Dissect.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  Graph in '..\Common\Graph.pas',
  SetUtils in '..\Common\SetUtils.pas',
  Assembly in '..\Common\Assembly.pas';

// Вспомогательная процедура удаления подмножеств внутри разбиений

procedure FreeSubSets(arg: TSet);
var  S : TSet;  // очередное разбиение (множество подмножеств)
begin
  S:= arg.GetFirst as TSet;   // выбор первого разбиения
  while Assigned(S) do begin
    S.ClrAndDestroy;          // удаление всех подмножеств разбиения
    S:= arg.GetNext as TSet;  // выбор следующего разбиения
  end;
end;

var
    Power   : integer;  // Мощность исходного множества
    Univers : TSet;     // Исходное множество объектов -- УНИВЕРСУМ
    Res: TSet;          // все разбиения множества

begin  {----- Main -----}
  repeat
    Write('Power= '); Readln(Power);  // Ввод мощности множества
    if Power<2 then Break;
```

```
    // Генерация множества символов заданной мощности
    Univers:= GenerateChars(Power);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');

    Res:= GenAllDissections(Univers);  // генерация всех разбиений
    Res.Expo;
    Writeln('- - - - - - - - - - - - -');

    FreeSubSets(Res);    // удаление подмножеств внутри всех разбиений
    Res.ClrAndDestroy;   // удаление самих разбиений
    Res.Free;            // удаление множества разбиений

    Univers.ClrAndDestroy; // очистка базового "универсума"
    Univers.Free;          // удаление базового "универсума"
  until false;
end.
```

Прежде всего, обратим внимание на глобальную переменную **Univers** — универсум. Она хранит исходное множество, разбиения которого надо получить. Важно помнить, что все создаваемые далее подмножества этого универсума будут содержать ссылки на его элементы (а не копии элементов). Стало быть, элементы универсума должны существовать, пока существуют подмножества этого универсума, то есть, универсум может быть уничтожен здесь лишь по завершении программы.

Примечание. Напомню, что *универсумом* обычно называют множество всех мыслимых объектов, но в контексте нашей программы мы сужаем это понятие.

Итак, после ввода требуемой мощности множества, мы создаём универсум из объектов-символов, вызывая функцию **GenerateChars** из модуля **SetUtils**. Затем формируем множество всех разбиений и выводим на экран. На этом можно было бы завершить программу, если бы наша система поддерживала автоматическую сборку «мусора». Такие системы существуют, но **Delphi** к ним не относится. Поэтому в реальных проектах от вас потребуется некоторая ловкость по части аккуратного освобождения памяти от ненужных объектов. Забывчивость в этом деле ведёт к засорению памяти, а преждевременное удаление объектов — к аварии программы. Поэтому я показываю здесь технику освобождения объектов, добавив ещё несколько строк.

В целом правило удаления объектов несложно: объекты удаляют в порядке, обратном их созданию. Универсум был создан первым, и будет удалён последним — тут всё ясно. Но над удалением результата **Res** надо подумать. Вызов деструктора через метод **Free** удалит лишь саму переменную-множество, но не его элементы, и тогда разбиения останутся в памяти. Предварительный вызов метода **ClrAndDestroy**, казалось бы, решает проблему — удаляет элементы-разбиения, но сами эти элементы тоже содержат множества (подмножества универсума), и они выживут, засорив память. Поэтому удаление надо начинать с них, с подмножеств, что и сделано у нас вспомогательной процедурой **FreeSubSets**. Отмечу, что это не единственный приём очистки, альтернативой

может быть конструирование на базе множества наследника, в деструкторе которого будут учтены тонкости его очистки.

В заключение приведу образец вывода нашей программой всех пяти разбиений трёхэлементного множества:

```
{ a b c } : 3 — исходное множество
- - - - - - - - - - - - -
{
  {
    { a b c } : 3
  } : 1

  {
    { a b } : 2
    { c } : 1
  } : 2

  {
    { b } : 1
    { a c } : 2
  } : 2

  {
    { a } : 1
    { b c } : 2
  } : 2

  {
    { a } : 1
    { b } : 1
    { c } : 1
  } : 3
} : 5
- - - - - - - - - - - - -
```

9.4. Случайные разбиения

Итак, наша программа генерирует *все* возможные разбиения множества. Однако научная ценность этого достижения сочетается — увы! — с практической его ничтожностью. Попытка получить все разбиения универсума всего лишь из восьми элементов посеет сомнения в мощи вашего компьютера. О крупных множествах и говорить нечего: тут вопрос будет лишь в том, что иссякнет раньше: память компьютера или ваше терпение.

Но кому интересны столь чудовищные множества разбиений? Ведь на практике мы имеем дело с разбиениями, мощность которых вполне разумна и составляет сотни, тысячи или миллионы. Такие разбиения формируются предикатами, продиктованными, как говорится, самой жизнью. Комбинируя разные предикаты и применяя их к разным подмножествам, можно получить как разбиения, так и пересекающиеся подмножества универсума (покрытия).

Но сейчас не о предикатах. Сейчас мы создадим на базе универсума некоторое количество его подмножеств с заданными характеристиками, причём сформируем

этот набор случайным образом. Результат этой работы пригодится нам в двух последующих главах, где мы займёмся обратной задачей — соединением подмножеств в целое.

Предстоящая работа будет сродни битью глиняных горшков. Представим себе некоторое количество одинаковых горшков (целый горшок — это универсум). Возьмём молоток, мешок, и, в сугубо научных целях, станем наполнять мешок черепками (количество их задано), поступая следующим образом. Разбив очередной горшок, мы отбросим случайно выбранную треть его обломков, а остальные две трети кинем в мешок. Горшки будут биться так, чтобы черепки получались и не слишком крупными, и не слишком мелкими, — эти граничные размеры осколков задаются параметрами программы. Вот и всё.

В последующих главах будем собирать из этих черепков целые горшки. Спрашивается: возможно ли это? В отношении реальных горшков — вряд ли. Но мы работаем с цифровыми «горшками», «черепки» которых дискретны, и потому при большом количестве обломков (от разных горшков) вполне вероятно склеить хотя бы один целый.

Сначала представлю вспомогательную функцию **GenRandSet** модуля **Dissect**, создающую одно случайное разбиение переданного ей множества. Можно сказать, что функция, как молоток, разбивает один горшок. Параметры **aMin** и **aMax** определяют минимальный и максимальный размеры «черепков».

Листинг 9-3 — Вспомогательная функция, генерирующая одно случайное разбиение

```
//  Генерация случайного разбиения исходного множества aSet

function GenRandSet(aMin, aMax: integer; aSet: TSet): TSet;
var n, k, j: integer;
    t: TItem;
    S: TSet;
begin
  Result:= CreateSet;                 // создаём множество-результат
  repeat
    n:= aMin + Random(aMax-aMin);     // случайная мощность подмножества
    S:= CreateSet;                    // создаём подмножество
    for j:= 1 to n do begin           // и заполняем случайным выбором элем-ов
      k:= 1 + Random(aSet.GetCount); // случайный индекс в исходном множестве
      t:= aSet.GetItem(k);            // случайный элемент исходного множества
      S.Insert(t);                    // вставляем в множество
      aSet.Delete(t);                 // и удаляем из исходного
      if aSet.GetCount=0 then Break; // выход из цикла, если множ. исчерпано
    end;
    if S.GetCount>0           // если множество не пусто,
      then Result.Insert(S)   // то вставляем в результат
      else S.Free;            // а иначе удаляем
  until aSet.GetCount=0;      // пока не исчерпано исходное множество
end;
```

Эта функция вызывается из другой — **GenSubsRand**, в результате работы которой создаётся буфер, заполненный подмножествами. Образно говоря, эта

функция создаёт мешок и складывает в него черепки. В качестве параметров передаются: исходное множество-универсум, количество генерируемых подмножеств и предельные размеры «черепков» в процентах от целого.

Листинг 9-4 — Функция GenSubsRand, генерирующая заданное количество случайных подмножеств

```
// Генерирует заданное в aCnt количество подмножеств исходного множества.
// Параметры задают минимальную и максимальную (%) мощность подмножеств

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;
var
  nMin, nMax: Integer;
  Copy: TSet;  // рабочая копия исходного множества aSet
  SS: TSet;    // очередное разбиение
  S : TSet;    // очередное множество внутри разбиения

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= TBuffer.Create;  // создаём буфер результата

  // Цикл создания заданного количества подмножеств

  while aCnt>0 do begin
    Copy:= aSet.Copy as TSet;   // создаём копию исходного множества
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);  // создаём случайное разбиение
      // Перебор подмножеств случайного разбиения:
      S:= SS.GetFirst as TSet;   // первое подмножество из разбиения
      while Assigned(S) and (aCnt>0) do begin
        if Random(3)<>0 then begin
          // копия очередного "осколка"
          // с вероятностью 2/3 вставляется в результат
          Result.Put(S.Copy);
          Dec(aCnt);
        end;
        S:= SS.GetNext as TSet;  // следующее подмножество из разбиения
      end;
      SS.ClrAndDestroy;  // очищаем разбиение
      SS.Free;           // и удаляем его
    until (aCnt=0) or (Copy.GetCount=0);
    Copy.Free;   // удаляем копию исходного множества
  end; // while
end;
```

Далее представлена тестирующая программа.

Листинг 9-5 — Программа для генерации случайных подмножеств

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
  Univers : TSet;      // Исходное множество объектов -- УНИВЕРСУМ
  Buf: TBuffer;      // Буфер с подмножествами

begin
    // создаём универсум из 20-ти объектов:
    Univers:= GenerateChars(20);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Формируем 10 подмножеств,
    // размер максимального "черепка" не превышает 40% от целого
    Buf:= GenSubsRand(Univers, 10, 0 {%}, 40 {%});
    Buf.Expo;
    Writeln('- - - - - - - - - - - - - -');
    Buf.ClrAndDestroy;
    Buf.Free;
    Univers.Free;
    Readln;
end.
```

Вот результат работы этой программы:

```
{ a b c d e f g h i j k l m n o p q r s t } : 20
- - - - - - - - - - - - - -
BufCnt= 10
{ d e g i k l o } : 7
{ s } : 1
{ b h j n q r t } : 7
{ c l m n o } : 5
{ f h r s } : 4
{ e g i } : 3
{ f l n t } : 4
{ c f j n p } : 5
{ k m q } : 3
{ h } : 1
- - - - - - - - - - - - - -
```

9.5. Разбиения из текстового файла

Модуль **Dissect** содержит ещё одну функцию — **GenSubsFromText**. Она генерирует псевдослучайные подмножества символов из текстового файла. Идея в том, чтобы, читая символ за символом, накапливать подмножество до тех пор, пока очередной символ не окажется «лишним», т.е. уже существующим в множестве. В этот момент накопление прерывается, подмножество заносится в буфер, а

накопитель очищается и всё повторяется. Одновременно формируется универсум — множество всех прочитанных из файла символов.

Листинг 9-6 — Функция создания подмножеств и универсума из файла

```
function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;
type TChrSet = set of char;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Формирование объекта-множества символов
    // из классического множества символов
    function MakeFromSet(const aSet: TChrSet): TSet;
    var c: char;
        t: TItemChar;
    begin
      Result:= CreateSet;
      for c in aSet do begin
        t:= TItemChar.Create(c);
        Result.Insert(t)
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var F: Text;            // текстовый файл
    S: string;          // строка файла
    c: char;            // очередной символ
    i: integer;         // индекс
    set1 : TChrSet;     // множество для УНИВЕРСУМа
    set2 : TChrSet;     // множество для подмножества
    subset: TSet;       // очередное подмножество
begin
  Result:= TBuffer.Create;
  set1:= [];
  set2:= [];
  Assign(F, aName); Reset(F);
  while not Eof(F) do begin
    if Result.GetCount >= aCnt then break;
    Readln(F, S);
    for i:= 1 to Length(S) do begin
      // Обработка очередной строки файла:
      if Result.GetCount >= aCnt then break;
      c:= Char(Ord(S[i]) or $20);       // low case (буквы 'a'..'z')
      if c in ['a'..'z'] then begin
        set1:= set1 + [c];              // для УНИВЕРСУМА
        if c in set2 then begin         // если символ встретился повторно
           subset:= MakeFromSet(set2);  // создаём подмножество
           Result.Put(subset);          // и заносим в результат
           set2:= [c];                  // начинаем следующее подмножество
        end else begin                  // если символ встретился впервые
           set2:= set2 + [c];           // расширяем текущее подмножество
        end;
      end;
    end;
  end;
  Close(F);
  subset:= MakeFromSet(set2);  // оставшееся множество
  Result.Push(subset);         // вставляем в результат
  aSet:= MakeFromSet(set1);    // и создаём универсум
end;
```

Работу функции можно проверить программой, подобной этой:

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
  Univers : TSet;      // УНИВЕРСУМ
  Buf: TBuffer;      // Буфер с подмножествами

begin
    Buf:= GenSubsFromText('Set_08c.dpr',  // имя файла
                          Univers,        // результат-УНИВЕРСУМ
                          10              // количество подмножеств
                         );
    Univers.Expo;
    Writeln('- - - - - - - - - - - - -');
    Buf.Expo;
    Buf.ClrAndDestroy;
    Buf.Free;
    Univers.Free;
    Writeln('- - - - - - - - - - - - -');
    Readln;
end.
```

9.6. Итоги

9.6.1. Разбиение множеств на подмножества происходит в ходе естественных процессов и умственной деятельности.

9.6.2. Подмножества одного *разбиения* в сумме дают исходное множество и взаимно не пересекаются.

9.6.3. Количество разбиений одного множества растёт с ростом его мощности по закону, обгоняющему экспоненту, и потому получить все возможные разбиения возможно только для небольших множеств.

9.6.4. Случайное разбиение множеств моделирует битьё горшков, такие разбиения будут использоваться в последующих главах.

9.7. Что почитать

№	Автор(ы)	Название	Главы, страницы
√ 1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
√ 8	Липский В.	Комбинаторика для программистов	Стр. 43-50
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 10
Задача о наименьшем разбиении (ЗНР)

10.1. Зачем собирают множества

В предыдущей главе мы разбивали множество на ряд подмножеств. Иногда подмножества рождаются как результат естественных процессов или умственной деятельности, а нам приходится решать обратную задачу: *собрать* нечто целое из этих «обломков». Вот пример. Пусть мастеру потребовался ряд слесарных инструментов, но продаются они в наборах. Тогда он будет искать самое дешёвое множество наборов, содержащее все нужные инструменты. Проблемы такого рода возникают сплошь и рядом. Так, некоторые задачи на графах сводятся к сборке множества из подмножеств. Ввиду важности этой проблемы мы уделим ей должное внимание в этой и следующей главе.

10.2. Разбиения и покрытия

Прежде, чем сформулировать «сборочные» задачи и перейти к их решению, определим два близких, но разных понятия: *разбиение* и *покрытие*, — и то, и другое являются множествами подмножеств. *Разбиение* — это такая совокупность взаимно *непересекающихся* подмножеств, из которых можно собрать исходное множество (универсум). *Покрытие* — это почти то же самое, но соединяемые подмножества могут взаимно *пересекаться* (накладываться).

Названия говорят сами за себя: *разбиение* — это результат битья посуды. Из осколков разных горшков можно собрать целый, но при этом нельзя заменить один черепок другим, если он, хотя бы чуточку, «напирает» на смежный. Иное дело — *покрытие*, которое сродни устиланию пола клочками некогда целых одинаковых ковров. Такие обрывки можно уложить с сохранением прежнего рисунка, допуская *наложение* отдельных совпадающих участков. Различие этих понятий демонстрирует рис. 10-1. Кстати, отметим, что *разбиение* множества является одновременно и его *покрытием*, но не наоборот.

10.3. ЗНР и ЗНП

Теперь сформулируем задачи *сборки* множества из его подмножеств. Пусть дана совокупность подмножеств некого универсума, — подобие мешка с черепками от битых горшков. В первом варианте *задачи о наименьшем разбиении* (ЗНР) требуется подобрать минимальное *количество взаимно непересекающихся* подмножеств, дающих в сумме исходный универсум. Во втором варианте этой же задачи каждому подмножеству назначается *цена*, и надо подобрать совокупность подмножеств с наименьшей общей *стоимостью*. Легко догадаться, что назначив всем «черепкам» единичную цену, первый вариант задачи можно решить через второй, поэтому всё внимание мы уделим решению второго варианта.

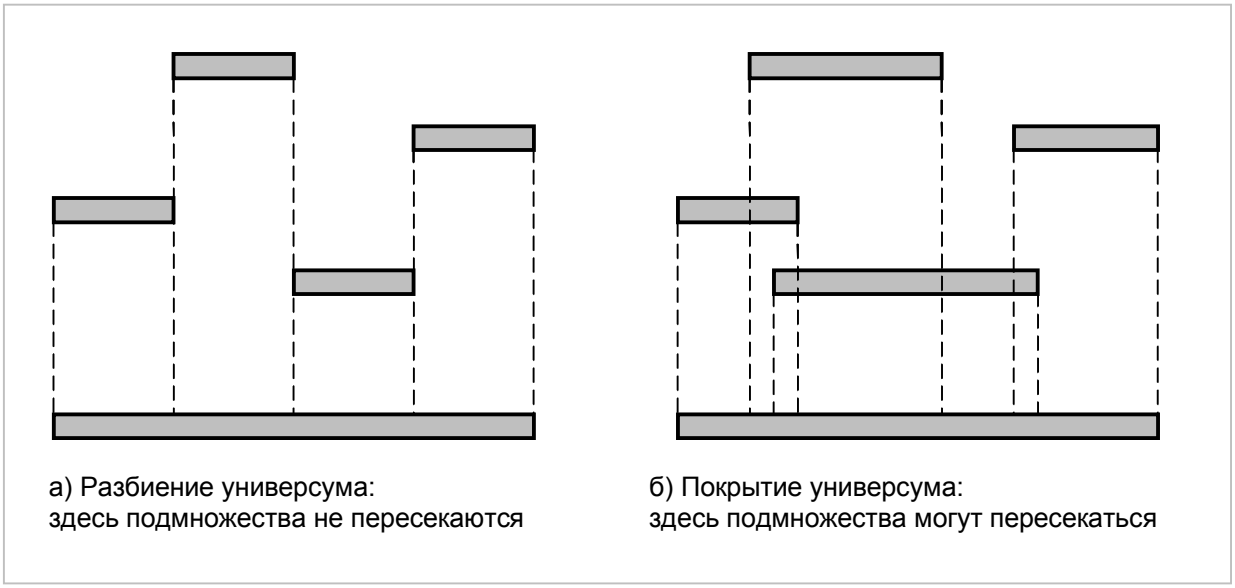


Рис. 10-1 — Разбиение (а) и покрытие (б)

Сходно ставятся и два варианта *задачи о наименьшем покрытии* (ЗНП). Дана та же совокупность подмножеств, — подобие фрагментов ковра. Надо найти либо наименьшее *количество* подмножеств, покрывающих универсум, либо множество подмножеств («лоскутов») с наименьшей *стоимостью*. Здесь не запрещается взаимное пересечение подмножеств. Так же, как и для ЗНР, первый вариант задачи будет частным случаем второго, который мы и будем решать.

На первый взгляд, кажется, что подобрать *покрытие* проще, чем *разбиение*, ведь требование взаимного непересечения подмножеств тут не предъявляется. Но дело обстоит ровно наоборот: количество возможных вариантов покрытий больше, и потому перебор этого «богатства» длится дольше. Компьютеру ЗНП даётся куда труднее ЗНР, в следующей главе вы убедитесь в этом.

10.4. Исходные данные

Решение ЗНР — задачи о наименьшем разбиении — начнём с описания исходных данных. Наш «мешок с черепками» представим набором подмножеств, где каждому подмножеству назначена цена. Нечто подобное мы генерировали случайным образом функциями **`GenSubsRand`** и **`GenSubsFromText`** (см. модуль **`Dissect`**). Но там не было цен. Теперь надо соединить каждое подмножество с числом, указывающим его цену. Здесь возможны разные пути, но мы объединим эти данные путём агрегатирования, то есть через создание нового класса **`TCostSet`**, содержащего оценку и множество в качестве своих полей. Класс унаследован от базового объекта **`TItem`**, ниже дано его объявление, а за подробностями отсылаю к модулю **`Root`**.

```
// Класс "множество с оценкой"

  TCostSet = class (TItem)
    protected
      procedure Clr(aDestroy: boolean);
    public
      mCost: integer;       // оценка, стоимость элемента
      mSet : TSet;          // подмножество
      mFlag  : boolean;     // флажок для пометок
      mDestroy : boolean;   // определяет способ уничтожения объекта
      constructor Create(aCost: integer; aSet: TSet; aDestroy : boolean);
      constructor CreateEmpty;
      destructor  Destroy;  override;
      function    Copy: TItem; override;
      procedure   Clear;
      procedure   ClrAndDestroy;
      function    Compare(arg: TItem): TCompare; override;
      procedure   Print(var aFile: TextFile); override;
      procedure   Append(arg : TCostSet);
      procedure   Insert(arg : TCostSet);
  end;
```

Итак, роль отдельного черепка исполняет объект класса **TCostSet**. А «мешком» послужит объект-буфер класса **TBuffer**. Две упомянутые выше функции генерируют такие буферы, наполненные подмножествами **TSet**, но нам требуется другое наполнение — объектами класса **TCostSet**. С этой целью в модуль **SetUtils** добавлена функция **GenCostItems**, преобразующая первый род буфера во второй. Наряду с исходным буфером, ей передаётся число, определяющее стратегию назначения цены элементам буфера, — возможность варьировать стратегией пригодится нам при испытании алгоритмов сборки множеств. Предусмотрены пять вариантов стратегии (см. листинг 10-1).

Листинг 10-1 — Функция формирования буфера подмножеств с ценами.

```
// Преобразует исходный набор элементов в набор элементов с ценами
// aBuf  - исходный набор
// aCost - стратегия оценки:
//    0 - случайные оценки в диапазоне от 1 до 99
//    1 - все оценки равны единице
//    2 - случайные оценки, пропорциональные мощности подмножества
//    3 - оценка равна мощности подмножества
//    4 - оценка пропорциональна квадрату мощности подмножества
function GenCostItems(aCost: integer; aBuf: TBuffer): TBuffer;
var S: TSet;
    cost: integer;
    n: integer;
begin
  Result:= TBuffer.Create;
  S:= aBuf.Get as TSet;
  while Assigned(S) do begin
    n:= S.GetCount;
    case aCost of
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
     else cost:= 1 + Random(99)
    end;
```

```
    Result.Put(TCostSet.Create(cost, S, true));
    S:= aBuf.Get as TSet;
  end;
end;
```

Стратегия «1» состоит в присвоении единичных цен всем подмножествам, — она нужна для решения первых вариантов ЗНР и ЗНП. Три следующие стратегии формируют цены, так или иначе зависящие от мощности подмножеств. Стратегия «0» и все прочие дают случайные цены в диапазоне от 1 до 99.

Ещё одной «кузницей» данных служит функция **ReadSetsFromText** (модуль **SetUtils**), — она читает данные из текстового файла, в котором цены уже указаны. В конце концов, обе функции нужны лишь для испытания создаваемых алгоритмов, за подробностями обращайтесь к листингу модуля **SetUtils**. Теперь переходим к обсуждению алгоритма ЗНР.

10.5. Предварительные соображения

10.5.1. Перебор

Предстоящее дело сродни сборке машины без чертежа, наобум. И тут ничего, кроме метода проб и ошибок, не придумать. Каков порядок сложности этой задачи? Пусть дана совокупность из **N** «осколков». Начнём с самой простой идеи. Взяв один из «черепков», что даёт **N** вариантов, попытаемся присоединить к нему любой из оставшихся **N-1**. Затем к этим двум — любой из оставшихся **N-2** и т.д. Тогда количество вариантов перебора определится формулой:

O(N) = N • (N-1) • (N-2) •...

Перед нами факториал, стало быть, задаче свойственна *экспоненциальная* сложность. Если действовать разумней, можно многократно уменьшить число комбинаций, и всё же сложность задачи останется *экспоненциальной*, и потому нам следует позаботиться ещё о двух вещах.

10.5.2. А существует ли решение?

Поскольку работа предстоит нешуточная, следует предварительно выяснить, а стоит ли за неё браться? Ведь никто не гарантирует, что предлагаемый набор подмножеств содержит хотя бы одно разбиение или покрытие. К счастью, проверить *недостижимость* решения очень легко и мы это обязательно сделаем.

10.5.3. Дубликаты

В связи с «крутизной» факториала каждый лишний элемент в наборе данных существенно усложняет задачу. А такие элементы при «битье горшков» вполне вероятны. Ведь разрушение наших виртуальных горшков даёт порой одинаковые «черепки», и тогда удаление дубликатов может снизить сложность на несколько порядков.

10.6. Возможно ли решение?

Это первое, что нас интересует. Ответ даёт функция **TestCoverOrDissect** (здесь и далее обсуждаем модуль **Assembly**). Идея состоит в последовательном объединении элементов исходного буфера до тех пор, пока мы либо получим требуемый универсум, либо исчерпаем буфер. Если не получилось накопить универсум, то решение предстоящих задач ЗНР и ЗНП невозможно (листинг 10-2).

Листинг 10-2 — Проверка возможности решения ЗНР и ЗНП

```
function TestCoverOrDissect(aUniv: TSet;      // универсум
                            aBuf: TBuffer  // буфер подмножеств
                           ): boolean;
var  CS : TCostSet;  // очередное подмножество с оценкой
    Sum : TSet;      // накопитель универсума
      i : integer;
begin
  Result:= false;
  Sum:= CreateSet;
  // Суммируем подмножества буфера до тех пор,
  // пока либо сумма достигнет универсума, либо исчерпается буфер
  for i:=1 to aBuf.GetCount do begin
    CS:= aBuf.GetByIndex(i) as TCostSet;
    Sum.Add(CS.mSet);
    if Sum.Compare(aUniv) in [cmpEq, cmpGreate] then begin
      Result:= True;
      Break;
    end;
  end;
  Sum.Free;
end;
```

Подчеркну, что хотя результат **TRUE** гарантирует существование хотя бы одного *покрытия*, он не гарантирует наличие *разбиения*.

10.7. Удаление дубликатов

Здесь на основе исходного буфера создадим новый, где каждый «черепок» будет уникальным. Если несколько элементов содержат одинаковые подмножества, то в результирующий буфер попадёт только один из этих элементов, причём самый «дешёвый» (листинг 10-3). Обратите внимание на то, что функция создаёт новый буфер, а исходный остаётся нетронутым.

Листинг 10-3 — Просеивание дубликатов большей или равной стоимости

```
function GenFilter(aBuf: TBuffer): TBuffer;
var Ci, Cj : TCostSet;
      i, j : integer;
      Comp : TCompare;
begin
  // Предварительный сброс флажков:
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    Ci.mFlag:= false;
  end;
```

```
  // Установка флажков для пропускаемых элементов
  // Отмечаем совпадающие подмножества с бОльшнй либо равной ценой
  for i:= 1 to aBuf.GetCount-1 do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    if Ci.mFlag then Continue;              // если обработан
    for j:= i+1 to aBuf.GetCount do begin
      Cj:= aBuf.GetByIndex(j) as TCostSet;
      if Cj.mFlag then Continue;            // если обработан
      Comp:= Ci.mSet.Compare(Cj.mSet);      // сравниваем подмножества
      if Comp = cmpEq then begin
        // Если сопадают, сравниваем цены:
        if Ci.mCost >= Cj.mCost
          then Ci.mFlag:= true    // i-й будет пропущен
          else Cj.mFlag:= true;   // j-й будет пропущен
      end;
    end;
  end;

  Result:= TBuffer.Create;
  // Перепись в буфер результата элементов со сброшенным флажком
  // Прокручиваем циклически исходный буфер aBuf
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.Get as TCostSet;
    aBuf.Put(Ci);
    // Если флаг не установлен, то заносим в буфер результата
    if not Ci.mFlag  then Result.Put(Ci);
  end;
end;
```

10.8. Подготовка блоков

Итак, убедившись в том, что решение ЗНР *вероятно* возможно, и отсеяв ненужные дубликаты, сосредоточимся на разумном переборе вариантов.

Воспользуемся тем, что *разбиение* может содержать лишь по одному элементу универсума (подмножества разбиения не должны пересекаться). Пусть множество-универсум составляют элементы **{ A,B,C,D... }**. Создадим изначально пустой накопитель универсума. Если взять любой «обломок», содержащий элемент **A** (и, вероятно, другие элементы), и присоединить его к накопителю универсума, то затем можно не рассматривать в качестве кандидатов иные подмножества, содержащие элемент **A**. Так мы отсекаем много заведомо негодных вариантов. Далее посмотрим, содержится ли в накопителе универсума элемент **B**? Если да, то не надо рассматривать «осколки», содержащие этот элемент, а если нет, то перебирать подмножества, содержащие элемент **B**, и не содержащие **A**. Точно так же поступим со всеми последующими элементами. Эти рассуждения наводят на мысль предварительно сгруппировать «черепки» следующим образом.

Пусть исходный набор данных — это «мешок с черепками». Возьмём несколько пустых коробок, количество которых совпадает с мощностью универсума (количеством элементов в нём), и пометим их именами элементов: **A,B,C** и т.д.. Затем станем раскладывать черепки из мешка в коробки. В первую сложим все обломки, содержащие элемент **A**, и тогда в мешке их больше не

останется. Во вторую коробку из оставшихся обломков сложим те, что содержат элемент **B**. Так же поступим с прочими обломками, пока мешок не опустеет, и в итоге все они окажутся разложенными в помеченных коробках. Обратите внимание на тенденцию: в последующих коробках количества черепков, как правило, уменьшается, вплоть до того, что некоторые из них останутся пустыми.

Но прежде, чем представить функцию, создающую множество упомянутых выше коробок, ознакомимся с объектом, представляющим собой эту коробку. У нас он будет объявлен и реализован классом «блок»:

Листинг 10-4 — Блок — класс множества для решения ЗНР и ЗНП

```
type
  TSetBlock = class (TSetList)
    mLabel : TItem;     // Метка блока
    constructor Create(aLabel: TItem);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: Text); override;
  end;

constructor TSetBlock.Create(aLabel: TItem);
begin
  inherited Create;
  mLabel:= aLabel;
end;

function TSetBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;  // порядок элементов совпадает с порядком вставки
end;

procedure TSetBlock.Print(var aFile: Text);
begin
  mLabel.Print(aFile);
  inherited;
end;
```

Обратите внимание: поскольку метод сравнения **Compare** всегда возвращает значение **cmpLess**, внутри множества блоки располагаются в том же порядке, в котором они вставлялись. Теперь покажем функцию, формирующую блоки (листинг 10-5).

Листинг 10-5 — Создание блоков для поиска разбиений

```
// aSet  - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)

function GenDissectBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из универсума
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet;                  // создаём множество-результат
  for i:=1 to aUniv.GetCount do begin  // цикл по элементам универсума
    if aBuf.GetCount=0 then Break;     // стоп при исчерпании буфера
```

```
    t:= aUniv.GetItem(i);         // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // и создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.Get as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)        // если оно содержит текущий элемент
        then Block.Insert(S)    // то помещаем подмножество S в блок
        else aBuf.Put(S);       // а иначе возвращаем подмножество в буфер
    end;
    if Block.GetCount>0         // если блок не пуст
      then Result.Insert(Block) // помещаем его в множество
      else Block.Free;          // а иначе удаляем пустой
  end;
  aBuf.Free;  // освобождаем буфер
end;
```

Покажем результаты создания вышеупомянутых наборов данных и блоков. Изначально универсум представляет собой множество из 15-ти элементов:

```
{ a b c d e f g h i j k l m n o } : 15
```

Вызов **`GenSubsRand(Univers,100,0,40)`** создал сотню представленных ниже случайных «осколков» этого универсума:

```
{ a } : 1
{ c d f g l } : 5
{ j m } : 2
{ d m } : 2
{ h j l o } : 4
{ a } : 1
{ g } : 1
{ i } : 1
{ k } : 1
{ b e h m } : 4
{ d i j k o } : 5
{ l } : 1
{ a c f g n } : 5
{ h o } : 2
{ d } : 1
{ e k } : 2
{ a f l } : 3
{ g i m } : 3
{ b } : 1
{ a } : 1
{ k } : 1
{ d i } : 2
{ f o } : 2
{ j n } : 2
{ c g } : 2
{ a l m o } : 4
{ d e g i n } : 5
{ o } : 1
{ b d i k l } : 5
{ h n } : 2
{ e f } : 2
{ j } : 1
{ i m } : 2
{ j } : 1
```

```
{ e } : 1
{ b f } : 2
{ h i l } : 3
{ d e j m o } : 5
{ a } : 1
{ c d k l n } : 5
{ g j } : 2
{ d f } : 2
{ m } : 1
{ i } : 1
{ b g k } : 3
{ h m } : 2
{ c e o } : 3
{ l } : 1
{ j } : 1
{ h i k l o } : 5
{ a d g j n } : 5
{ b m } : 2
{ e f } : 2
{ c } : 1
{ g l n } : 3
{ c d m } : 3
{ a g j m } : 4
{ i } : 1
{ c h n } : 3
{ f k } : 2
{ a } : 1
{ f l o } : 3
{ j } : 1
{ d e h l o } : 5
{ a c m } : 3
{ g } : 1
{ b f i k n } : 5
{ e i j } : 3
```

```
{ f l } : 2
{ d } : 1
{ b c f h i } : 5
{ a g j k } : 4
{ l o } : 2
{ c i l n } : 4
{ b j k } : 3
{ f o } : 2
{ a d e g h } : 5
{ m } : 1
{ b g } : 2
{ d n } : 2
{ a j } : 2
{ h m } : 2
{ c } : 1
{ a b d h m } : 5
{ j } : 1
{ b m } : 2
{ a f h i } : 4
{ d e l n o } : 5
{ c g k } : 3
{ c g k } : 3
{ h i } : 2
{ o } : 1
{ b d l } : 3
{ e } : 1
{ c d j k o } : 5
{ e f h i m } : 5
{ g } : 1
{ b l } : 2
{ f g h i l } : 5
{ e j m n } : 4
```

Вызов функции **`GenCostItems(0, SubsBuf)`** создал набор тех же подмножеств со случайными ценами:

```
Cost=   6  { a } : 1
Cost=  10  { c d f g l } : 5
Cost=  36  { j m } : 2
Cost=  88  { d m } : 2
Cost=  21  { h j l o } : 4
Cost=  94  { a } : 1
Cost=  59  { g } : 1
Cost=  98  { i } : 1
Cost=  79  { k } : 1
Cost=  39  { b e h m } : 4
Cost=   3  { d i j k o } : 5
Cost=   3  { l } : 1
Cost=  61  { a c f g n } : 5
Cost=  78  { h o } : 2
Cost=  47  { d } : 1
Cost=  51  { e k } : 2
Cost=  74  { a f l } : 3
Cost=  93  { g i m } : 3
Cost=  28  { b } : 1
Cost=  16  { a } : 1
Cost=  14  { k } : 1
Cost=  99  { d i } : 2
Cost=  14  { f o } : 2
Cost=  34  { j n } : 2
Cost=  16  { c g } : 2
Cost=  42  { a l m o } : 4
Cost=  50  { d e g i n } : 5
Cost=  11  { o } : 1
Cost=  38  { b d i k l } : 5
Cost=  30  { h n } : 2
Cost=  28  { e f } : 2
Cost=   7  { j } : 1
Cost=  30  { i m } : 2
Cost=  84  { j } : 1
Cost=  24  { e } : 1
Cost=  38  { b f } : 2
Cost=  58  { h i l } : 3
Cost=   6  { d e j m o } : 5
Cost=  27  { a } : 1
Cost=  40  { c d k l n } : 5
Cost=   8  { g j } : 2
Cost=  76  { d f } : 2
Cost=  42  { m } : 1
Cost=   3  { i } : 1
Cost=  71  { b g k } : 3
Cost=  52  { h m } : 2
Cost=  44  { c e o } : 3
Cost=  72  { l } : 1
Cost=   1  { j } : 1
Cost=  10  { h i k l o } : 5
Cost=  24  { a d g j n } : 5
Cost=  53  { b m } : 2
```

```
Cost=  80  { e f } : 2
Cost=  62  { c } : 1
Cost=  27  { g l n } : 3
Cost=  83  { c d m } : 3
Cost=  48  { a g j m } : 4
Cost=  46  { i } : 1
Cost=   5  { c h n } : 3
Cost=  67  { f k } : 2
Cost=  67  { a } : 1
Cost=  43  { f l o } : 3
Cost=  24  { j } : 1
Cost=  58  { d e h l o } : 5
Cost=  35  { a c m } : 3
Cost=  10  { g } : 1
Cost=  85  { b f i k n } : 5
Cost=  58  { e i j } : 3
Cost=  84  { f l } : 2
Cost=   3  { d } : 1
Cost=  67  { b c f h i } : 5
Cost=  67  { a g j k } : 4
Cost=  84  { l o } : 2
Cost=  36  { c i l n } : 4
Cost=  82  { b j k } : 3
Cost=  80  { f o } : 2
Cost=  91  { a d e g h } : 5
Cost=   8  { m } : 1
Cost=  27  { b g } : 2
Cost=  51  { d n } : 2
Cost=  74  { a j } : 2
Cost=  78  { h m } : 2
Cost=  45  { c } : 1
Cost=  39  { a b d h m } : 5
Cost=  17  { j } : 1
Cost=  89  { b m } : 2
Cost=  47  { a f h i } : 4
Cost=  14  { d e l n o } : 5
Cost=  92  { c g k } : 3
Cost=  62  { c g k } : 3
Cost=  59  { h i } : 2
Cost=   9  { o } : 1
Cost=  45  { b d l } : 3
Cost=   4  { e } : 1
Cost=  70  { c d j k o } : 5
Cost=  14  { e f h i m } : 5
Cost=  20  { g } : 1
Cost=  10  { b l } : 2
Cost=  35  { f g h i l } : 5
Cost=  62  { e j m n } : 4
```

После отсева дубликатов в наборе осталось 87 элементов, из которых были сформированы блоки, указанные в табл. 10-1 (в скобках указаны мощности блоков).

Табл. 10-1 — Разбиение набора подмножеств на блоки

Метка	Блок	Метка	Блок
a **(12)**	Cost= 24 { a d g j n } : 5 Cost= 6 { a } : 1 Cost= 39 { a b d h m } : 5 Cost= 42 { a l m o } : 4 Cost= 35 { a c m } : 3 Cost= 47 { a f h i } : 4 Cost= 48 { a g j m } : 4 Cost= 61 { a c f g n } : 5 Cost= 67 { a g j k } : 4 Cost= 91 { a d e g h } : 5 Cost= 74 { a f l } : 3 Cost= 74 { a j } : 2	*h* **(7)**	Cost= 10 { h i k l o } : 5 Cost= 21 { h j l o } : 4 Cost= 30 { h n } : 2 Cost= 58 { h i l } : 3 Cost= 52 { h m } : 2 Cost= 59 { h i } : 2 Cost= 78 { h o } : 2
b **(12)**	Cost= 10 { b l } : 2 Cost= 38 { b d i k l } : 5 Cost= 39 { b e h m } : 4 Cost= 67 { b c f h i } : 5 Cost= 27 { b g } : 2 Cost= 45 { b d l } : 3 Cost= 85 { b f i k n } : 5 Cost= 38 { b f } : 2 Cost= 71 { b g k } : 3 Cost= 53 { b m } : 2 Cost= 82 { b j k } : 3 Cost= 28 { b } : 1	*i* **(2)**	Cost= 3 { i } : 1 Cost= 30 { i m } : 2
c **(10)**	Cost= 5 { c h n } : 3 Cost= 10 { c d f g l } : 5 Cost= 40 { c d k l n } : 5 Cost= 16 { c g } : 2 Cost= 36 { c i l n } : 4 Cost= 70 { c d j k o } : 5 Cost= 44 { c e o } : 3 Cost= 62 { c g k } : 3 Cost= 83 { c d m } : 3 Cost= 45 { c } : 1	*j* **(3)**	Cost= 1 { j } : 1 Cost= 34 { j n } : 2 Cost= 36 { j m } : 2
d **(10)**	Cost= 3 { d i j k o } : 5 Cost= 6 { d e j m o } : 5 Cost= 14 { d e l n o } : 5 Cost= 3 { d } : 1 Cost= 50 { d e g i n } : 5 Cost= 58 { d e h l o } : 5 Cost= 51 { d n } : 2 Cost= 76 { d f } : 2 Cost= 88 { d m } : 2 Cost= 99 { d i } : 2	*k* **(1)**	Cost= 14 { k } : 1
e **(6)**	Cost= 14 { e f h i m } : 5 Cost= 4 { e } : 1 Cost= 28 { e f } : 2 Cost= 62 { e j m n } : 4 Cost= 58 { e i j } : 3 Cost= 51 { e k } : 2	*l* **(2)**	Cost= 3 { l } : 1 Cost= 84 { l o } : 2
f **(5)**	Cost= 35 { f g h i l } : 5 Cost= 14 { f o } : 2 Cost= 43 { f l o } : 3 Cost= 67 { f k } : 2 Cost= 84 { f l } : 2	*m* **(1)**	Cost= 8 { m } : 1
g **(4)**	Cost= 8 { g j } : 2 Cost= 27 { g l n } : 3 Cost= 10 { g } : 1 Cost= 93 { g i m } : 3	*o* **(1)**	Cost= 9 { o } : 1

Табл. 10-1 интересна в следующем смысле. Во-первых, мы не видим здесь блока с меткой «**n**», поскольку все подмножества с этим элементом оказались в предшествующих блоках. Во-вторых, обратим внимание на порядок следования

подмножеств внутри каждого блока, он определён методом **TCostSet.Compare**, за полным текстом которого отсылаю в модуль **Root**. Этот метод ранжирует подмножества внутри блока в порядке возрастания *удельной стоимости* его элементов, что выражено следующим оператором:

```
if mCost * (arg as TCostSet).mSet.GetCount <
   (arg as TCostSet).mCost * mSet.GetCount
  then  Result:= cmpLess
  else  ...
```

Здесь при сравнении удельных стоимостей трудоёмкое деление с плавающей точкой заменено быстрым умножением целых чисел. То есть, вместо

(A/B) < (C/D)

сравниваются

(A•D) < (C•B)

В итоге подмножества в блоках расположились в порядке возрастания удельной стоимости элементов. Так, первым трём элементам первого блока присущи следующие удельные стоимости:

```
24/5 = 4.8
 6/1 = 6.0
39/5 = 7.8
```

Существен ли порядок блоков при полном переборе? Поскольку от полного перебора мы уйдём, прибегнув к *поиску с возвратом*, то чем скорее будет найдено минимальное разбиение, тем большая часть заведомо непригодных ветвей будет отсечена. Поскольку элементы перебираются в естественном порядке, размещение «дешёвых» вариантов в начале блока повышает шансы раннего обнаружения минимального разбиения.

10.9. Поиск с возвратом

Итак, распределив осколки по коробкам, мы тем самым на много порядков снизили сложность задачи. Теперь максимальное количество возможных вариантов перебора выражается произведением мощностей блоков, что для нашей таблицы даёт число **1.45E+08**. Не так уж мало для скромного набора исходных данных, и это подвигает нас к дальнейшему улучшению алгоритма — применению *поиска с возвратом*, с которым мы познакомились в главе 8.

Поскольку целью подбора является минимальная стоимость разбиения, мы примем текущую накопленную стоимость критерием, управляющим возвратом. Учредим две переменные: в первой будем накапливать текущую стоимость промежуточных (неполных) разбиений, а во второй — хранить стоимость лучшего разбиения, найденного на текущий момент. Изначально, когда накопленный универсум пуст, текущая стоимость равна нулю. Стоимость наилучшего разбиения

пока неизвестна, и для правильной работы алгоритма мы занесём туда заведомо большое число (условную бесконечность **`MaxInt`**).

Теперь приступим к перебору подмножеств, начав с первого блока. Взявши блок, прежде всего, проверим, содержится ли метка блока в накопленном универсуме. Если да, то блок пропускаем и переходим к следующему. Понятно, что первый блок нам не проскочить, и мы переберём все его элементы. Взяв очередное подмножество, смотрим, не будет ли его стоимость вкупе с уже накопленной меньше минимальной. При поиске первого из возможных разбиений это условие всегда выполняется, и мы проверяем следующее условие: пересекается ли очередное подмножество с накопленным универсумом. Если нет, то присоединяем его к универсуму и включаем в разбиение, а затем проверяем полноту накопленного универсума. Если накопленный универсум полон (совпадает с заданным в параметре функции), то текущее разбиение и его стоимость запоминаются в качестве лучших на данный момент. Если же накопитель универсума не полон, переходим к обработке следующего блока, и так вплоть до последнего.

Поскольку все блоки обрабатываются одинаково, есть смысл применить рекурсию. Функция, реализующая вышеописанный алгоритм, представлена ниже (листинг 10-6). В основном теле сначала проверяется возможность решения: если оно невозможно, работа завершается. Затем удаляются дубликаты, создаются блоки и несколько вспомогательных переменных, впоследствии вызывается локальная рекурсивная процедура обработки блоков, которая и выполняет основную работу.

Отметим, что даже когда предварительная оценка исходного набора подмножеств оставляет шанс найти разбиение, оно, тем не менее, может и не существовать, и тогда функция вернёт пустое множество. Если же равноценных разбиений окажется несколько, функция вернёт первое найденное.

Листинг 10-6 — Решение задачи о наименьшем разбиении (ЗНР)

```
function CollectMinDissect(aUniv: TSet;      // целевой универсум
                           aBuf: TBuffer   // буфер с подмножествами
                           ): TCostSet;
var Buf     : TBuffer;  // Отфильтрованная копия входного буфера
    Univ    : TSet;       // Накопитель универсума
    BestCost: integer;    // Лучшая стоимость
    Blocks  : TSet;       // Вспомогательные блоки
    Res : TCostSet;       // Исходное пустое разбиение
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aCost: integer;   // накопленная сумма
                        aSum: TSet;       // накопленные элементы
                        aRes: TCostSet    // накопленные подмножества
                        );
  var Block: TSetBlock; // текущий блок
      Sum : TSet;       // Накопленное множество элементов
      Res : TCostSet;   // Накопленное разбиение (множество подмножеств)
       CS : TCostSet;   // очередное оценённое подмножество из блока
  begin
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Если элемент текущего блока содержится в накопителе
    if aSum.Exist(Block.mLabel) then begin
      // то пропускаем этот блок и входим в следующий (если не последний)
      if aBlock < Blocks.GetCount then begin
        BlockHandle(aBlock+1, aCost, aSum, aRes) // вход в следующий блок
      end
    end else begin
      // перебор подмножеств текущего блока
      CS:= Block.GetFirst as TCostSet;
      {$ifdef _TRACE_} TestCount:= TestCount+Block.GetCount;{$endif}
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        // Если сумма не превышена
        // и CS не пересекается с накопл. универсумом, то обрабатываем
        if ((aCost+CS.mCost) < BestCost) and
           not aSum.TestIntersect(CS.mSet) then begin
          // Пристраиваем к разбиению очередной блок:
          Sum:= aSum.Copy as TSet;     // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного разбиения
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // пристраиваем к разбиению очередной блок
          // Проверяем, накоплен ли целевой универсум
          if Sum.GetCount = aUniv.GetCount then begin
            // Здесь найдено очередное разбиение.
            // Поскольку оно лучше прежнего,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;   // новая лучшая сумма
            Result.Free;                   // удаляем прежнее разбиение
            Result:= Res.Copy as TCostSet; // и создаём новое лучшее
            {$ifdef _TRACE_ -- трассировка }
            Inc(TestNumber);  // порядковый номер результата
            Writeln(TestNumber:3, BestCost:7, aBlock:8);
            {$endif}
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res)
          end; // else
```

```
          // Перед выбором следующего подмножества в блоке освобождаем:
          // возвращаем из стека предыдущие значения
          Res.Free;    // накопленное разбиение
          Sum.Free;    // накопленный универсум
        end; // if
        CS:= Block.GetNext as TCostSet;
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin

  Result:= TCostSet.CreateEmpty;  // Создаём пустое множество-разбиение

  // Проверяем возможность найти разбиение, иначе выход

  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf);  // Создаём фильтрованную копию входного буфера
  {$ifdef _TRACE_ -- трассировка }
    Writeln('--- Collect Min Dissect ---');
    // Печать результата фильтрации:
    Writeln('aBuf / Buf = ', aBuf.GetCount, ' / ', Buf.GetCount);
  {$endif}

  Blocks:= GenDissectBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  //Blocks.Expo;
  {$ifdef _TRACE_ -- трассировка }
    // Для статистики:
    Writeln('Num  BestCost  Block');
    TestNumber:= 0;                    // порядковый номер решения
    TestCount:= 0;                     // количество просмотренных элементов
    TestDiff:= CalcDifficult(Blocks);  // теоретическая сложность
  {$endif}
  // Подготовка переменных:
  BestCost:= MaxInt;          // Лучшая сумма
  Univ := CreateSet;          // Накопитель универсума
  Res:= TCostSet.CreateEmpty; // Создаём пустое множество-разбиение

  // Рекурсивная обработка блоков:
  BlockHandle(1, 0, Univ, Res);

  // Очистка:
  Res.Free;                   // Исходное пустое разбиение
  Univ.Free;                  // Освобождаем накопитель
  Blocks.ClrAndDestroy;       // Удаляем блоки
  Blocks.Free;                // и множество блоков

  {$ifdef _TRACE_ -- трассировка }
     Writeln('TestDiff  = ', TestDiff :11);
     Writeln('TestCount = ', TestCount :11);
     if TestCount>0
      then Writeln('Diff/Count= ', TestDiff / TestCount :11);
  {$endif}

end;
```

10.10. Тестирование и статистика

Для тестирования решения ЗНР служит следующая программа.

Листинг 10-7 — Программа для тестирования ЗНР

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
    Univers : TSet;       // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;        // Результат - минимальное разбиение

begin
    // Создание универсума из 15-ти элементов
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
    Writeln('- - - - - - - - - - - - - -');
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;  // исхлдный буфер уже не нужен
    CostBuf.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Формирование минимального разбиения (решение ЗНР):
    Res:= CollectMinDissect(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;
    CostBuf.Free;
    Univers.ClrAndDestroy;
    Univers.Free;
    Write('OK');  Readln;
end.
```

Результат работы программы выводится в следующем виде:

```
Cost= 70
{
 Cost=   3    { d i j k o } : 5
 Cost=   5    { c h n } : 3
 Cost=  10    { b l } : 2
 Cost=   6    { a } : 1
 Cost=   8    { m } : 1
 Cost=  10    { g } : 1
 Cost=  28    { e f } : 2
} : 7
```

В первой строке показана стоимость разбиения, а в последующих — множество его подмножеств. Здесь можно вывести и дополнительную статистику, если в модуле **Assembly** определить имя **_TRACE_** так, как показано ниже:

```
{$define _TRACE_}
```

Тогда в ходе поиска разбиения будет выведена дополнительная информация в следующем виде:

```
aBuf / Buf = 100 / 76
Num  BestCost  Block
  1    200       9
  2    116      14
  3     70      13
TestDiff  =  1.45E+0008
TestCount =  1.03E+0003
Diff/Count=  1.41E+0005
```

Вначале показано количество элементов в буфере до и после фильтрации дубликатов. Затем тремя колонками выводятся номер найденного разбиения, его стоимость и номер блока, где оно обнаружено. В конце приводятся три числа:

- **TestDiff** — теоретическая сложность, вычисленная через произведение мощностей блоков (количество вариантов перебора);
- **TestCount** — количество реально обработанных подмножеств;
- **Diff/Count** — соотношение этих чисел (коэффициент выигрыша).

Интересно проследить зависимость трудоёмкости задачи от стратегии назначения цен (см. табл. 10-2). В таблице приведены данные для множества из 15-ти элементов и массива из 76 подмножеств, оставшихся после фильтрации дубликатов. Здесь видно, что компьютеру легче «расправиться» с элементами, цена которых задана случайно. Чуть сложнее даётся массив с одинаковыми ценами, а трудней всего такой, где цена совпадает с мощностью подмножества (стратегия 3).

Табл. 10-2 — Влияние ценовой стратегии

Параметр стратегии	Стратегия (Cost=)	TestDiff	TestCount	Diff/Count
0	1 + Random(99)	1.45E+08	1.03E+03	1.41E+05
1	1	1.45E+08	3.92E+03	3.71E+04
2	1 + (n div 2) + Random(n)	1.45E+08	1.63E+04	8.89E+03

3	**n**	**1.45E+08**	**5.75E+04**	**2.52E+03**
4	**1 + (n-1)*(n-1)**	**1.45E+08**	**2.69E+04**	**5.39E+03**

Разумеется, что приведенные выше статистические оценки следует воспринимать лишь как грубый ориентир, более точное сравнение доступно лишь после испытаний с засечкой времени.

10.11. Итоги

10.11.1. К сборке универсума из подмножеств сводятся многие практически важные задачи.

10.11.2. Множество подмножеств, в сумме дающих универсум, является либо *разбиением* (отдельные его подмножества не пересекаются), либо *покрытием* (подмножества могут пересекаться). Разбиение является частным случаем покрытия.

10.11.3. Особую ценность имеют *минимальные* разбиения и покрытия, причём задача может ставиться двояко: ищется либо минимальное *количество* подмножеств, либо набор с минимальной *стоимостью*. Первый вариант задачи решается через второй путём назначения подмножествам единичных цен.

10.11.4. Задачи поиска минимальных разбиений (ЗНР) и покрытий (ЗНП) обладают экспоненциальной сложностью по отношению к мощности универсума.

10.11.5. Для снижения сложности задач в исходном наборе подмножеств удаляют дубликаты, затем его *разбивают на блоки* и применяют *поиск с возвратом*.

10.11.6. Избежать заведомо невыполнимой работы можно предварительной проверкой набора данных на предмет невозможности собрать покрытие или разбиение.

10.12. Что почитать

№	Автор(ы)	Название	Главы, страницы
√ 1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 53
√ 8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 11
Задача о наименьшем покрытии (ЗНП)

Продолжим сборку множеств, перейдя к поиску наименьших *покрытий* (ЗНП). Рассмотрим пример, где капитан судна нанимает экипаж, выбирая среди матросов, каждый из которых владеет одной или несколькими профессиями. Если требование компактности экипажа первично, он постарается набрать минимум матросов так, чтобы покрыть все нужды в специалистах. Если же матросы требуют для себя разные уровни оплаты, а цель капитана — экономия средств, то он наберёт по возможности «дешевый» экипаж, покрывающий, однако, всю потребность в профессионалах. Этим двум целям соответствуют два варианта задачи о наименьшем покрытии: 1) найти минимальное *количество* подмножеств покрытия, и 2) найти набор подмножеств с *минимальной общей стоимостью*.

Решение ЗНП очень похоже на поиск минимального разбиения (ЗНР): здесь тоже предварительно выясняют возможность покрытия, а также удаляют ненужные дубликаты подмножеств. Но на последующих шагах учитывают одно отличие: подмножества, входящие в покрытие, могут пересекаться, а это влияет и на формирование блоков, и на их перебор.

11.1. Составление блоков

Обратимся к образу покрытия как совокупности ковровых лоскутов. При «реставрации» ковра обрывки можно частично накладывать друг на друга, пересекать. И потому содержимое известных вам меченых коробок с лоскутами должно быть несколько иным. В первую сложим *все* клочки, в состав которых входит элемент **A**, во вторую — *все* клочки с элементом **B**, и т.д. Но, позвольте, как это возможно? Ведь клочок может попасть только в одну из коробок. Если клочок содержит и **A,** и **B**, то он попадёт только в коробку с меткой **A**. Да, с реальными лоскутами так не сделаешь. Но мы работаем с информацией, и указатели на объекты позволят нам этот фокус.

Итак, предварительная подготовка меченых блоков даст нам множество «коробок», число которых совпадает с мощностью универсума. Если встречаемость элементов в разных подмножествах равновероятна, то и мощность блоков будет примерно одинакова (вспомним, что для ЗНР она в целом убывала). Всё потому, что одно подмножество вставляется сразу в несколько блоков, и это же влечёт чудовищную трудоёмкость перебора в ЗНП. Отметим, что так же, как и в ЗНР, подмножества внутри блоков расположим в порядке не убывания их удельной стоимости (дешёвые в начале), а это повышает вероятность раннего обнаружения наилучшего покрытия.

В следующем листинге представлена функция, генерирующая множество меченых блоков для поиска наименьшего покрытия.

Листинг 11-1 — Создание блоков для поиска минимального покрытия

```
function GenCoverBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из aSet
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet; // создаём буфер результата
  for i:=1 to aUniv.GetCount do begin // Цикл по элементам универсума
    t:= aUniv.GetItem(i);         // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.GetByIndex(j) as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)       // если оно содержит текущий элемент
        then Block.Insert(S)  // то помещаем подмножество S в блок
    end;
    Result.Insert(Block) // вставляем блок в результирующее множество
  end;
end;
```

Ниже показаны четыре из пятнадцати сформированных блоков:

```
a  {
 Cost=  24      { a d g j n } : 5
 Cost=   6      { a } : 1
 Cost=  39      { a b d h m } : 5
 Cost=  42      { a l m o } : 4
 Cost=  35      { a c m } : 3
 Cost=  47      { a f h i } : 4
 Cost=  48      { a g j m } : 4
 Cost=  61      { a c f g n } : 5
 Cost=  67      { a g j k } : 4
 Cost=  91      { a d e g h } : 5
 Cost=  74      { a f l } : 3
 Cost=  74      { a j } : 2
  } : 12

 b  {
 Cost=  10      { b l } : 2
 Cost=  38      { b d i k l } : 5
 Cost=  39      { a b d h m } : 5
 Cost=  39      { b e h m } : 4
 Cost=  67      { b c f h i } : 5
 Cost=  27      { b g } : 2
 Cost=  45      { b d l } : 3
 Cost=  85      { b f i k n } : 5
 Cost=  38      { b f } : 2
 Cost=  71      { b g k } : 3
 Cost=  53      { b m } : 2
 Cost=  82      { b j k } : 3
 Cost=  28      { b } : 1
  } : 13

 c  {
 Cost=   5      { c h n } : 3
 Cost=  10      { c d f g l } : 5
 Cost=  40      { c d k l n } : 5
 Cost=  16      { c g } : 2
 Cost=  36      { c i l n } : 4
 Cost=  35      { a c m } : 3
 Cost=  61      { a c f g n } : 5
 Cost=  67      { b c f h i } : 5
 Cost=  70      { c d j k o } : 5
 Cost=  44      { c e o } : 3
 Cost=  62      { c g k } : 3
 Cost=  83      { c d m } : 3
 Cost=  45      { c } : 1
  } : 13

 d  {
 Cost=   3      { d i j k o } : 5
 Cost=   6      { d e j m o } : 5
 Cost=  10      { c d f g l } : 5
 Cost=  14      { d e l n o } : 5
 Cost=   3      { d } : 1
 Cost=  24      { a d g j n } : 5
 Cost=  38      { b d i k l } : 5
 Cost=  39      { a b d h m } : 5
 Cost=  40      { c d k l n } : 5
 Cost=  50      { d e g i n } : 5
 Cost=  58      { d e h l o } : 5
 Cost=  70      { c d j k o } : 5
 Cost=  45      { b d l } : 3
 Cost=  91      { a d e g h } : 5
 Cost=  51      { d n } : 2
 Cost=  83      { c d m } : 3
 Cost=  76      { d f } : 2
 Cost=  88      { d m } : 2
 Cost=  99      { d i } : 2
  } : 19
```

11.2. Поиск с возвратом

Перебор подмножеств внутри блоков здесь очень похож на перебор при поиске разбиения. Создаются те же вспомогательные переменные, и точно так же перебираются подмножества: «по горизонтали» — внутри блока, и «по вертикали» — рекурсивным входом в следующий блок. Если накопитель универсума уже содержит какой-то элемент, то блок, меченный этим элементом, пропускается. Единственное отличие состоит в отсутствии проверки на пересечение очередного подмножества с накопленным универсумом (листинг 11-2).

Листинг 11-2 — Задача о наименьшем покрытии (ЗНП)

```
function CollectMinCover(aUniv: TSet;       // целевой универсум
                         aBuf: TBuffer   // буфер подмножеств с оценками
                        ): TCostSet;
var Buf: TBuffer;       // отфильтрованный входной буфер
    Univ    : TSet;       // Накопленный универсум
    BestCost: integer;    // Лучшая сумма
    Blocks  : TSet;       // Вспомогательные блоки
    Res : TCostSet;       // Исходное пустое покрытие
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блоков
```

```
  procedure BlockHandle(aBlock, aCost: integer; aSum: TSet; aRes: TCostSet);
  var Block : TSetBlock; // текущий блок
      Sum : TSet;        // Накопленный универсум
      Res : TCostSet;    // Накопленное покрытие (множество подмножеств)
       CS : TCostSet;    // очередное оценённое подмножество из блока
  begin
    // Выбираем блок из множества блоков
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Проверяем наличие метки текущего блока в накопителе универсума
    if aSum.Exist(Block.mLabel) then begin
      // Метка уже есть, пропускаем данный блок
      // и входим в следующий, если текущий не последний
      if aBlock<Blocks.GetCount then BlockHandle(aBlock+1,aCost,aSum,aRes)
    end else begin
      // Начало обработки очередного блока
      CS:= Block.GetFirst as TCostSet; // первое подмножество блока
      // Перебор подмножеств блока
      {$ifdef _TRACE_} TestCount:= TestCount+Block.GetCount;{$endif}
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        if (aCost+CS.mCost) < BestCost then begin // сумма меньше ?
          Sum:= aSum.Copy as TSet;      // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного покрытия
          Sum.Add(CS.mSet);      // накапливаем универсум
          Res.Insert(CS);        // и покрытие
          // Проверяем, накоплен ли целевой универсум (покрытие):
          if Sum.GetCount = aUniv.GetCount then begin
            // Найдено очередное покрытие,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее покрытие
            Result:= Res.Copy as TCostSet; // и сохраняем копию нового
            {$ifdef _TRACE_ -- трассировка }
            Inc(TestNumber);  // порядковый номер очередного решения
            Writeln(TestNumber:3, BestCost:7, aBlock:8);
            {$endif}
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res);
          end; // else
          // Перед выбором следующего подмножества из блока удаляем:
          Res.Free;  // накопленное покрытие
          Sum.Free;  // накопленный универсум
        end; // then
        CS:= Block.GetNext as TCostSet;  // следующий элемент блока
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin
  Result:= TCostSet.CreateEmpty; // Создаём пустое покрытие
  // Проверяем, если покрытие недостижимо, то выход:
  if not TestCoverOrDissect(aUniv, aBuf) then Exit;
  Buf:= GenFilter(aBuf); // Создаём фильтрованную копию входного буфера
  {$ifdef _TRACE_ -- трассировка }
    Writeln('--- Collect Min Cover ---');
    // Печать результата фильтрации:
    Writeln('aBuf / Buf = ', aBuf.GetCount, ' / ', Buf.GetCount);
  {$endif}
  Blocks:= GenCoverBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  Buf.Free;                             // Буфер уже не нужен, удаляем
```

```
  {$ifdef _TRACE_ -- трассировка }
    // Для статистики:
    Writeln('Num  BestCost  Block');
    TestNumber:= 0;                    // порядковый номер решения
    TestCount:= 0;                     // количество просмотренных элементов
    TestDiff:= CalcDifficult(Blocks);  // теоретическая сложность
  {$endif}
  Res:= TCostSet.CreateEmpty; // Начальное пустое покрытие
  Univ := CreateSet;          // Накопитель элементов (универсум)
  BestCost:= MaxInt;          // Лучшая сумма
  BlockHandle(1, 0, Univ, Res);  // Рекурсивная обработка блоков
  // Очистка:
  Res.Free;             // Начальное пустое покрытие
  Univ.Free;            // удаляем накопитель элементов (универсум)
  Blocks.ClrAndDestroy; // удаляем блоки
  Blocks.Free;
 {$ifdef _TRACE_ -- трассировка }
    Writeln('TestDiff  = ', TestDiff :11);  // теоретическая сложность
    Writeln('TestCount = ', TestCount:11);  // фактическая сложность
    // Соотношение
    if TestCount>0 then Writeln('Diff/Count= ', TestDiff / TestCount :11);
 {$endif}
end;
```

11.3. Тестирование

Для тестирования ЗНП применена следующая программа:

Листинг 11-3 — Программа для тестирования ЗНП

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
    Univers : TSet;       // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;        // Результат - минимальное покрытие
begin
    // Создание универсума
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;  // исхлдный буфер уже не нужен
    // Формирование минимального покрытия:
    Res:= CollectMinCover(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;    CostBuf.Free;
    Univers.ClrAndDestroy;    Univers.Free;
    Write('OK');
    Readln;
end.
```

Если в модуле **Assembly** определить имя **_TRACE_**, это повлечёт вывод дополнительной статистики (операторы вывода статистики в листингах не показаны, обратитесь к тексту модуля). Так получены следующие результаты:

```
{ a b c d e f g h i j k l m n o } : 15
- - - - - - - - - - - - - - -
--- Collect Min Cover ---
aBuf / Buf = 100 / 76   -- число элементов до и после удаления дубликатов
Num  BestCost  Block
  1     58       9
  2     56      11
  3     40       6
TestDiff  =  3.53E+0017
TestCount =  9.43E+0002
Diff/Count=  3.74E+0014
Cost= 40
{
 Cost=   3    { d i j k o } : 5
 Cost=   6    { d e j m o } : 5
 Cost=   5    { c h n } : 3
 Cost=  10    { c d f g l } : 5
 Cost=  10    { b l } : 2
 Cost=   6    { a } : 1
} : 6
```

В сравнении этих результатов с полученными при поиске разбиений (ЗНР) бросается в глаза чудовищное количество возможных комбинаций: **`3.53E+17`** против **`1.45E+08`**. Интересно исследовать влияние «ценовой политики» на трудоёмкость задачи, в табл. 11-1 дан результат этих испытаний.

Табл. 11-1 — Влияние ценовой стратегии

Параметр стратегии	Стратегия (Cost=)	TestDiff	TestCount	Diff/Count
0	1 + Random(99)	3.53E+17	9.43E+02	3.74E+14
1	1	3.53E+17	3.62E+04	9.75E+12
2	1 + (n div 2) + Random(n)	3.53E+17	2.06E+05	1.71E+12
3	n	3.53E+17	7.06E+06	5.00E+10
4	1 + (n-1)*(n-1)	3.53E+17	1.06E+06	3.32E+11

Наибольшая трудоёмкость присуща линейной зависимости цены от мощности подмножества (параметр 3), здесь программа мечется, что той герой в известной интермедии про раков, выбирающий между мелкими по три и крупными по пять. Но вчера.

11.4. Градиентный поиск

Предварительные оценки показывают (а будущие испытания это подтвердят), что поиск наилучшего покрытия может недопустимо затянуться. Как тут быть системам реального времени, где нужны быстрые решения? Можно ли выиграть время, частично пожертвовав качеством решения? По отношению к покрытию это значит, что мы согласны принять не самый дешёвый набор подмножеств, пусть он будет немного дороже идеала, главное — быстро.

Применённый далее подход известен как *жадный*, он часто применяется в задачах перебора. Применительно к ЗНП его иногда называют *градиентным поиском* или *градиентным спуском*. Последнее название напоминает о стратегии

горнолыжника, который, не зная трассы, стремится быстрее спуститься с вершины к подножью. В каждой точке трассы он выбирает направление с наибольшим уклоном, поддерживая, таким образом, максимальную скорость. Это, впрочем, не гарантирует ему рекордного результата, поскольку может увеличить длину трассы, что повлечёт потерю времени. Однако что ещё остаётся горе-лыжнику? Его подход не лишён смысла, и мы им воспользуемся.

Прежде всего, отметим, что подготовка блоков для ЗНП остаётся в силе, но обработка их изменится. Мы не станем многократно «нырять» и «выныривать» из блоков, как это делалось в рекурсивной процедуре: блоки будут обработаны всего лишь за один проход «слева направо» и «сверху вниз».

Взяв очередной блок, сначала проверим, содержится ли его метка в накопителе универсума? Если да, то блок игнорируем и переходим к следующему. Если нет, то ищем в блоке одно единственное подмножество, добавление которого к накопителю даст наилучший эффект. Для этого вычислим разность между очередным подмножеством и универсумом и определим удельную стоимость этой прибавки. Так, перебрав все подмножества блока, мы найдём подмножество, дающее наибольшую прибавку, отнесённую к единице его стоимости, — это подмножество и добавим к накопителю.

Функция градиентного поиска, дающая быстрое, хоть и не всегда идеальное решение ЗНП, представлена ниже.

Листинг 11-4 — Градиентный поиск покрытия

```
function CollectGradCover(aBase: TSet;
                          aBuf: TBuffer
                         ): TCostSet;
var    Buf : TBuffer;  // отфильтрованный входной буфер
      Univ : TSet;       // накопитель универсума
    Blocks : TSet;       // блоки
     Block : TSetBlock;  // текущий блок
    BestCS : TCostSet;   // текущее лучшее покрытие
  //-------------------------------------------------------------
  function GetBest: TCostSet;
  var      CS : TCostSet;   // очередное подмножество с оценкой
            S : TSet;       // рабочее подмножество
    BestDelta : integer;    // лучшее приращение
     BestCost : integer;    // лучшая цена
            i : integer;
  begin
    Result:= nil;
    S:= CreateSet;  // создаём рабочее множество
    BestDelta:=0;   // лучшая мощность
    BestCost:= 1;   // лучшая цена
    // перебор элементов блока:
    for i:= 1 to Block.GetCount do begin
      CS:= Block.GetItem(i) as TCostSet;  // очередное подмнож. с оценкой
      S.CopyItems(CS.mSet);               // копируем подмножество
      S.Sub(Univ);                        // оставляем приращение множества
      // Если данное приращение лучшее, то запоминаем
      if S.GetCount * BestCost > CS.mCost * BestDelta then begin
        Result:= CS;             // лучшее множество
        BestCost:= CS.mCost;     // лучшая цена
```

```
        BestDelta:= S.GetCount;  // лучшее приращение
      end;
    end;
    S.Free;  // удаляем рабочее множество
  end;
  //------------------------------------------------------------

var    i : Integer;

begin
  Result:= TCostSet.CreateEmpty;  // Покрытие = пустое множество
  // Если покрытие невозможно, вернуть пустой результат
  if not TestCoverOrDissect(aBase, aBuf) then Exit;

  Buf:= GenFilter(aBuf); // Отфильтрованная копия входного буфера
  Blocks:= GenCoverBlocks(aBase, Buf);  // Вспомогательные блоки
  Buf.Free;                      // Буфер уже не нужен
  Univ:= CreateSet;              // Создаём накопитель универсума
  for i:= 1 to Blocks.GetCount do begin
    Block:= Blocks.GetItem(i) as TSetBlock;
    if Univ.Exist(Block.mLabel)  // если метка блока найдена в накопителе
      then Continue;             // то пропускаем блок
    BestCS:= GetBest;            // выбор лучшего подмножества в блоке
    if Assigned(BestCS) then begin
      // Если лучшее подмножество в блоке найдено:
      Univ.Add(BestCS.mSet);     // накапливаем универсум
      Result.Insert(BestCS);     // и покрытие
    end;
  end;
  Univ.Free;              // освободить накопитель универсума
  Blocks.ClrAndDestroy;   // очистить вспомогательные блоки
  Blocks.Free;
end;
```

Для проверки функции, и сравнения градиентного поиска с точным решением ЗНП служит следующая программа:

Листинг 11-5 — Программа для сравнения двух методов поиска покрытия

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
    Univers : TSet;       // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;        // Результат - минимальное покрытие
begin
    // Создание универсума
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - -');
    // Генерация случайных разбиений универсума
    SubsBuf:= GenSubsRand(Univers, 100, 0, 40);
```

```
    {
     Добавление к подмножествам оценок,
     Первый параметр определяет способ назначения оценок:
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(1, SubsBuf);
    SubsBuf.Free;  // исходный буфер уже не нужен
    // Точное формирование минимального покрытия:
    Res:= CollectMinCover(Univers, CostBuf);
    Res.Expo;
    Res.Free;
    Writeln('- - - - - - - - - - - - -');
    // Быстрое формирование минимального покрытия:
    Res:= CollectGradCover(Univers, CostBuf);
    Res.Expo;
    // Очистка памяти:
    Res.Free;
    CostBuf.ClrAndDestroy;     CostBuf.Free;
    Univers.ClrAndDestroy;     Univers.Free;
    Write('OK');
    Readln;
end.
```

Отметим, что «ценовая политика», так сильно влияющая на скорость поиска в ЗНП, никак не влияет на градиентный поиск.

11.5. Временные испытания

Теперь сравним быстродействие решённых нами задач поиска минимальных разбиений и покрытий так, как это делают в спорте: засечкой времени. Тем мы учтём все операции. Тестирующая программа представлена ниже.

Листинг 11-6 — Программа для сравнения времени решений ЗНР и ЗНП

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Dissect in '..\Common\Dissect.pas',
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var
    Univers : TSet;       // Универсум - исходное множество объектов
    SubsBuf : TBuffer;  // Исходное множество подмножеств (без оценок)
    CostBuf : TBuffer;  // Буфер подмножеств с оценками
    Res: TCostSet;        // Результат - минимальное покрытие
    Start, Time : TDateTime;

begin
```

```
    Univers:= GenerateChars(15);
    Univers.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - - - -');
    // Генерация разбиений
    SubsBuf:= GenSubsRand(Univers, 150, 0, 40);
    // Добавление к подмножествам оценок
    {
     0: cost:= 1 + Random(99);
     1: cost:= 1;
     2: cost:= 1 + (n div 2) + Random(n);
     3: cost:= n;
     4: cost:= 1 + (n-1)*(n-1);
    }
    CostBuf:= GenCostItems(0, SubsBuf);
    SubsBuf.Free;

    Start:= Now;
    Res:= CollectMinDissect(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - -');

    Start:= Now;
    Res:= CollectMinCover(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - -');

    Start:= Now;
    Res:= CollectGradCover(Univers, CostBuf);
    Time:= MilliSecondsBetween(Start, Now);
    Res.Expo; Res.Free;
    Writeln('Time (ms) = ', Time:6:0);
    Writeln('- - - - - - - - - - - - - - - - - - - - -');
    CostBuf.Free;
    Univers.ClrAndDestroy;    Univers.Free;
    Write('OK');
    Readln;
end.
```

Исходный набор содержит 150 «осколков», из которых после отсева дубликатов остаются 115. Для пяти ценовых стратегий были получены результаты, представленные в табл. 11-2 и табл. 11-3 .

Табл. 11-2 — Время поиска разбиений (ЗНР) и покрытий (ЗНП), ms

Параметр стратегии	Стратегия (Cost=)	ЗНР	ЗНП	
			Точное решение	Градиентный поиск
0	1 + Random(99)	15	31	16
1	1	31	124	15
2	1 + (n div 2) + Random(n)	406	2124	16
3	n	7’015	171’957	15
4	1 + (n-1)*(n-1)	2’327	22’219	15

Табл. 11-3 — Стоимость найденных разбиений и покрытий

Параметр стратегии	Стратегия (Cost=)	ЗНР	ЗНП	
			Точное решение	Градиентный поиск
0	1 + Random(99)	64	44	57
1	1	4	4	4
2	1 + (n div 2) + Random(n)	12	12	13
3	n	15	15	15
4	1 + (n-1)*(n-1)	15	15	15

Абсолютное время зависит от мощи компьютера, но сравнение времён показательно. Из первой таблицы видно, что разбиение ищется существенно быстрее покрытия, поскольку блоки в ЗНР содержат меньше элементов. Сравнение двух методов ЗНП выводит в чемпионы градиентный поиск. А что у чемпиона с качеством? Из табл. 11-3 видно, что в трёх случаях из пяти градиентный поиск дал идеальный результат, в одном случае — всего на 8% хуже (13 против 12), и ещё однажды он уступил на 30% (57 против 44). Разумеется, что исчерпывающие выводы преждевременны, но градиентный поиск заслуживает внимания.

11.6. Комбинированный поиск

Итак, мы решили ЗНП двумя способами: первый точный, но долгий, а второй быстрый, но неточный. Можно ли в системах реального времени объединить выгоды того и другого? Вспомним, что точное решение ЗНП даёт ряд промежуточных результатов, постепенно приближающихся к идеалу. Тогда есть смысл сначала быстро найти градиентный результат, а затем дать ход точному перебору, ограничив его требуемым интервалом времени. Если за это время перебор не завершится, то в качестве результата взять лучшее из двух: либо градиентный, либо последний промежуточный результат.

11.7. Итоги

11.7.1. Задачи поиска наименьшего покрытия (ЗНП) похожи на задачи поиска наименьших разбиений (ЗНР). Отличие состоит в том, что подмножества, составляющие покрытие, могут взаимно пересекаться.

11.7.2. При формировании блоков для решения ЗНП каждое подмножество включается во все блоки, помеченные его элементами.

11.7.3. При переборе подмножеств внутри блока не проверяется взаимное пересечение подмножества-кандидата с накопленным универсумом.

11.7.4. Ввиду большого количества элементов в блоках, решение ЗНП может быть недопустимо долгим. Это время сильно зависит также от цен подмножеств.

11.7.5. Альтернативное решение ЗНП — чрезвычайно быстрый градиентный поиск, не гарантирующий, однако, точного результата.

11.7.6. В системах, где время на поиск решения ограничено, возможно комбинирование градиентного поиска с точным перебором.

11.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
√ 1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 53
√ 8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 12
Внутреннее представление графа

Этой главой открывается этап повествования, предметом которого станут *графы*. Теория графов получила развитие с тех пор, как математики и философы догадались, что для понимания природы важны не только (и даже не столько) свойства объектов, сколько их междоусобные отношения. Не потому ли нас увлекают детективы, где сыщик докапывается до тайных связей между предметами, событиями и людьми? Теперь и мы примерим к себе эту завидную роль, ведь задачи на графах не уступают детективным дознаниям ни в сложности, ни в увлекательности!

Граф можно представить рисунком, наглядно показывающим *связи* между объектами. Простейшую связь создают *два* объекта, — такие связи называют *бинарными*. Иногда связь (рефлексивная) замыкается на единственном объекте, но всё равно считается бинарной. Обычно объект связан с несколькими другими, однако, каждая такая связь в отдельности тоже бинарная. Предметом нашего интереса будут графы, в которых между парой объектов существует *не более одной* связи — такие графы называют *простыми*.

Впрочем, простота их обманчива. Даже такими графами можно моделировать массу разнообразных отношений. Связи между объектами могут быть и симметричными, и направленными. Связи могут обладать рядом качеств: весом, длиной, пропускной способностью. К тому же и связываемые объекты (вершины графа) обычно наделены какими-то свойствами. В пример можно привести сеть дорог, где соединяемые города характеризуются населённостью, а дороги — длиной и количеством полос. Всё это в совокупности с разнообразием «узоров», образуемых связями, порождает море интересных и практически важных задач. В этой главе мы ступим первый шаг к их решению: перенесём граф-картинку в память компьютера.

12.1. С точки зрения математиков

Математики определяют граф G как упорядоченное множество, составленное из двух множеств, и записывают это так:

$$G = < X, A >$$

где X — множество вершин графа, A — множество его рёбер (или дуг — для ориентированного графа). Угловые скобки показывают, что множество упорядочено, иначе говоря, множество вершин указано первым. Эта формальная запись привлекает своим лаконизмом, но не годится для компьютерных манипуляций с компонентами графа. Нам предстоит разработать структуру, пригодную для эффективного представления графа в памяти компьютера.

12.2. Матричные представления

В книгах по дискретной математике обычно приводят матричные (табличные) представления графа, а именно: матрицу *инциденций* и матрицу *смежности*.

12.2.1. Матрица инциденций

На рис. 12-1 показан орграф с пронумерованными дугами, а в табл. 12-1 — матрица его инциденций.

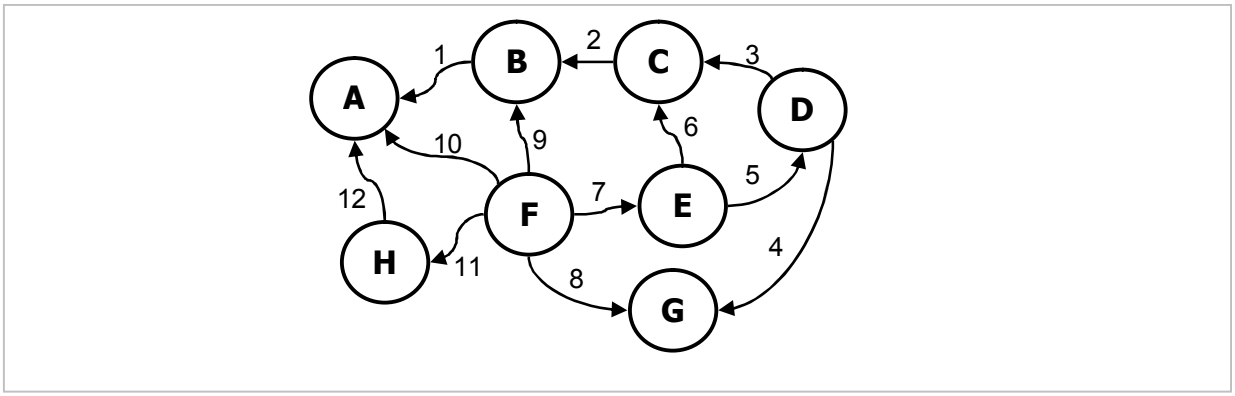


Рис. 12-1 — Пример ориентированного графа

В строках таблицы отмечены вершины, а в столбцах — дуги. Каждая непустая клетка таблицы содержит либо единицу, если дуга выходит из данной вершины, либо минус единицу, если дуга входит в неё. Пустые ячейки означают отсутствие дуг между вершинами. Вместо единиц можно указать числа, характеризующие, к примеру, расстояния между вершинами. Так или иначе, но каждый столбец содержит всего пару чисел, сумма которых равна нулю.

Табл. 12-1 — Матрица инциденций

Вершины	Дуги (номера дуг)											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
A	-1									-1		-1
B	1	-1							-1			
C		1	-1			-1						
D			1	1	-1							
E					1	1	-1					
F							1	1	1	1	1	
G				-1				-1				
H											-1	1

Очевидно, что ввиду разреженности таблицы почти вся отведённая ей память будет пустовать, а это не понравится ни программисту, ни компьютеру.

12.2.2. Матрица смежности

В левом столбце матрицы смежности (табл. 12-2) указаны вершины, из которых исходят дуги, а в шапке — вершины, в которые они входят. В клетках единицами или числами, означающими вес дуги, отмечены соседние вершины.

Табл. 12-2 — Матрица смежности

Исходящие вершины	Входящие вершины							
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A								
B	1							
C		1						
D			1				1	
E			1	1				
F	1	1			1		1	1
G								
H	1							

Из таблицы легко представить граф зрительно. Так, например, по строке *A* видно, что из этой вершины невозможно попасть в другие, то же относится к строке *G*. Зато вершина *F* соседствует сразу с пятью вершинами: *A*, *B*, *E*, *G*, *H*.

По столбцу можно найти вершины, из которых непосредственно достижима данная вершина. Например, вершина *A* достижима из вершин *B*, *F* и *H*.

Число, стоящее на главной диагонали, означает рефлексивную связь вершины с самой собой (в данной таблице таких связей нет). Общее число непустых клеток в матрице смежности равно количеству дуг графа.

Как и в матрице инциденций, вместо единиц можно указать числа, характеризующие длину (вес) дуги.

Матричные представления наглядны и удобны для математиков, но не всегда устраивают программистов. Причина — неэффективное использование памяти. Представим матрицей взаимные знакомства людей. Круг знакомств большинства граждан составляют не более сотни человек: родственники, соседи, сотрудники. Для жителей деревни почти все клетки в небольшой таблице смежности будут заполнены, — это нормально. Однако для населения крупного города или страны таблица будет очень велика, и большинство её клеток окажутся пустыми.

Экономного использования памяти можно добиться на разреженных матрицах, но сам по себе этот механизм непрост. Мы пойдём иным путём: представим графы *множествами*, — такой приём будет сочетать эффективность с удобством формальных математических описаний.

12.3. Представление множествами

12.3.1. Основные идеи

Вернёмся к матрице смежности, и «выжмем» из неё лишь непустые клетки. Так для некоторой вершины *X* сформируем два множества соседних вершин: тех, что достижимы из неё (в строке), и тех, из которых она достижима (в столбце). На рис. 12-2 показан пример этих множеств для вершины *C*.

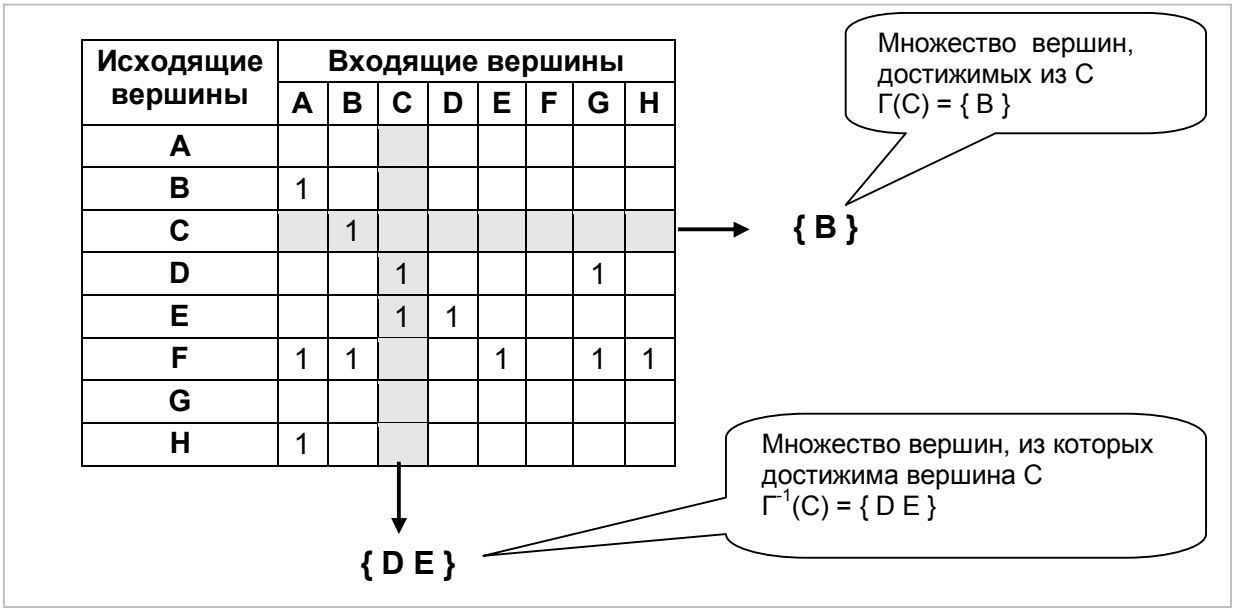


Рис. 12-2 — Множества достижимостей для вершины C

Отметим, что для полноценного представления графа достаточно одного из двух множеств, — другое находится алгоритмически. Но пара множеств значительно ускоряет многие алгоритмы на орграфе. В неориентированном графе будем применять только одно множество, поскольку в нём оба множества совпадают.

В табл. 12-3 тот же орграф представлен совокупностью множеств. В левом столбце даны вершины, а в двух соседних — множества смежных им вершин.

Табл. 12-3 — Представление орграфа множествами

Множество вершин	Множество Г	Множество $Г^{-1}$
A	{ }	{ B F H }
B	{ A }	{ C F }
C	{ B }	{ D E }
D	{ C G }	{ E }
E	{ C D }	{ F }
F	{ A B E G H }	{ }
G	{ }	{ D F }
H	{ A }	{ F }

12.3.2. Терминология

Теперь условимся о терминах. Смежные вершины часто обозначают греческой буквой *Г* — «гамма». Будем называть *исходящей гаммой* вершины *X*, или просто *гаммой*, множество *достижимых из неё* соседей. Обозначать исходящую гамму вершины *X* будем так:

Г(X) или $Г^{1}(X)$

Множество смежных вершин, *из которых* достижима данная вершина *X* назовём *входящей гаммой* и обозначим так:

$Г^{-1}(X)$ — гамма степени минус один (входящая).

Следуя далее этому принципу, введём обозначения, показанные в табл. 12-4.

Табл. 12-4 — Обозначение исходящих и входящих гамм

Гаммы		**Пояснение**
Исходящие	**Входящие**	
$Г^{0}(X)$	$Г^{0}(X)$	**Сама исходная вершина** *X*
$Г^{1}(X)$	$Г^{-1}(X)$	**Соседи вершины** *X*
$Г^{2}(X)$	$Г^{-2}(X)$	**Соседи соседей вершины** *X*
$Г^{3}(X)$	$Г^{-3}(X)$	**Соседи соседей соседей вершины** *X*

Для примера на рис. 12-3 показаны исходящие и входящие гаммы вершины *D*.

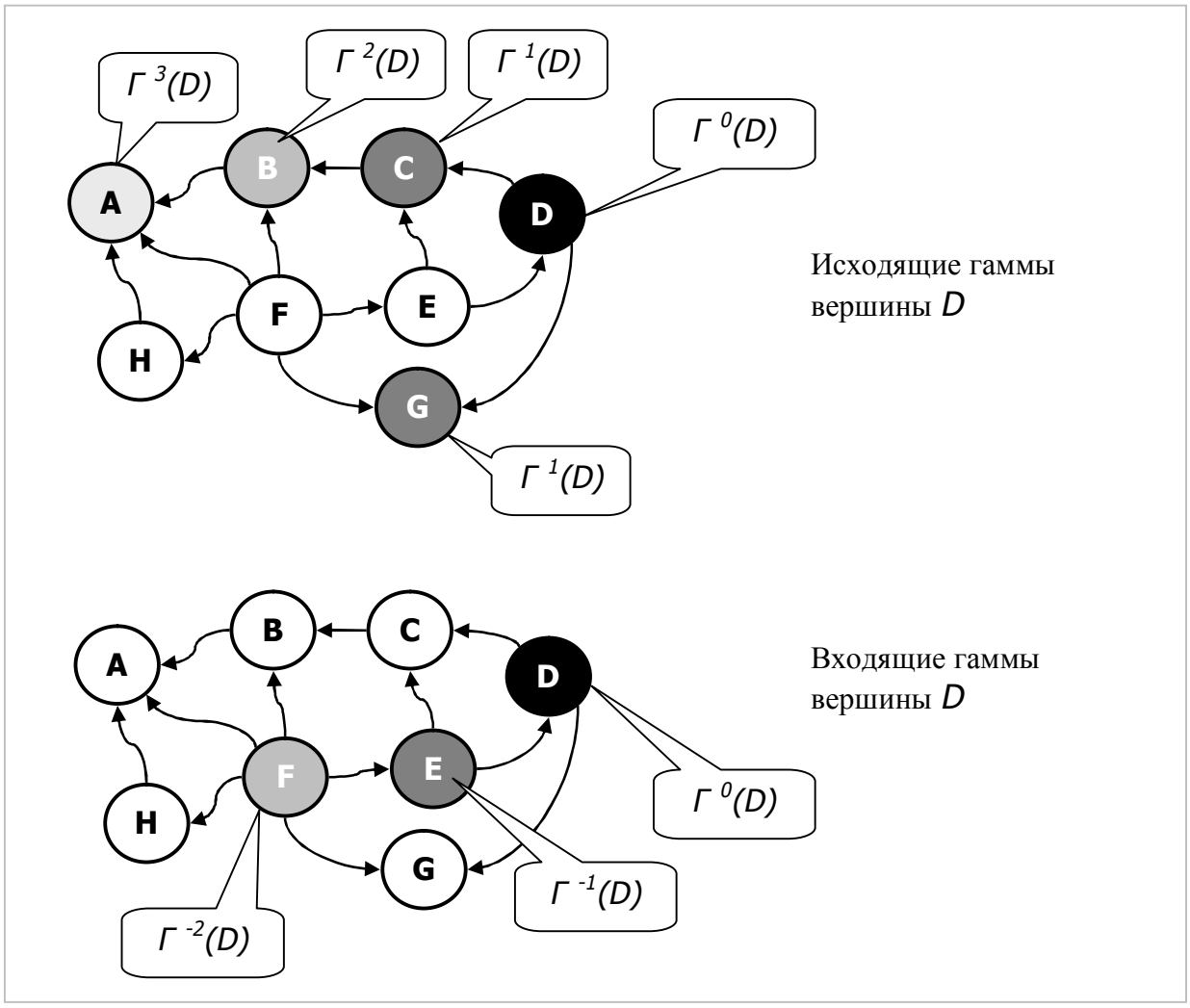


Рис. 12-3 — Исходящие и входящие гаммы вершины D

Иными словами, гамма степени ноль — это сама вершина, а все последующие *исходящие* гаммы степеней **N** доступны из неё за **N** шагов. По отношению к *входящим* гаммам степени **N** можно сказать, что эта вершина доступна из них за **N** шагов.

Объединив все исходящие гаммы вершины *X*, получим *интегральную* исходящую гамму — множество вершин, которые доступны из *X*. Аналогично объединением входящих гамм получим *интегральную* входящую гамму — множество вершин, из которых достижима вершина *X*. Далее «гаммой» будем называть множества достижимых вершин; точный смысл термина будет ясен из контекста, либо дополнительно пояснён.

12.3.3. Граф и его составляющие

Для представления графа в памяти компьютера нам потребуются элементы трёх классов: собственно граф **TGraph** и две его составляющие: вершины **TNode** и связи **TLink** (линки). Все они будут потомками базового класса **TItem** (элемент).

Рассмотрим основные информационные поля этих объектов, а к методам обратимся по мере их разработки. Информационные поля графа таковы:

```
TGraph = class (TItem)     // Граф в целом
    protected
      mName : String;      // Произвольная строка, отображаемая при выводе
      mDirect : boolean;   // Признак ориентированного графа
      mNodes : TSet;       // Множество вершин графа
      . . .
    public
      mLoadNodes : boolean;  // Признак нагруженных вершин
      mLoadLinks : boolean;  // Признак нагруженных связей
      . . .
  end;
```

Этим набором полей можно представить любые разновидности графов: ориентированные и не ориентированные, с нагруженными и ненагруженными вершинами и связями.

Класс **TNode** описывает вершину графа:

```
  TNode = class (TItem)       // Вершина (узел) графа
    private
      mLnkOut : TSet;         // Множество исходящих связей
      mLnkIn  : TSet;         // Множество входящих связей
    protected
      mOwner  : TGraph;       // Ссылка на содержащий узел граф
      mColor  : integer;      // Цвет (используется в алгоритмах)
      mPred   : TNode;        // Предшествующий узел (Дейкстра)
      mDist   : integer;      // Расстояние (Дейкстра)
      . . .
    public
      mValue : integer;       // Нагрузка узла
      . . .
  end;
```

Поля **mLnkOut** и **mLnkIn** — это множества смежных вершин данной вершины, иначе говоря, это исходящая и входящая гаммы первой степени. Для неориентированного графа используем только **mLnkOut**. Поле **mOwner** — это ссылка на граф, содержащего эту вершину, через это поле методы вершины получают доступ к полям графа. Вспомогательные поля **mColor**, **mPred**, **mDist** нужны методам, реализующим стандартные алгоритмы на графах. Поле **mValue** задаёт «вес» вершины в тех задачах, где он учитывается. Другие поля, содержащиеся в вершине, будем пояснять по мере решения соответствующих задач.

Теперь покажем связи (линки) — это элементы множеств **mLnkOut** и **mLnkIn**, содержащихся в вершине **TNode**. Структура элемента связи **TLink** такова:

```
TLink = class (TItem)
    protected
      mLoadLinks : boolean;   // Признак нагруженных связей
```

```
    . . .
  public
    mValue: integer;  // вес дуги (стоимость, расстояние)
    mOwner: TNode;    // вершина-источник дуги (владелец)
    mDest : TNode;    // вершина-приёмник дуги
    . . .
end;
```

Поле **mValue** задаёт «вес» дуги (длину или стоимость пути). Поля **mOwner** и **mDest** определяют две вершины: источник и приёмник дуги соответственно.

Для отражения каждой связи *ориентированного* графа создаётся *один* линк, но вставляется он в двух местах: в множество **mLnkOut** узла-источника, и в множество **mLnkIn** узла-приёмника.

В *неориентированном* графе каждая связь представлена *двумя* линками: прямым и обратным; они вставляются в множества **mLnkOut** вершины-источника и вершины-приёмника. В этом типе графа поле **mLnkIn** не используется.

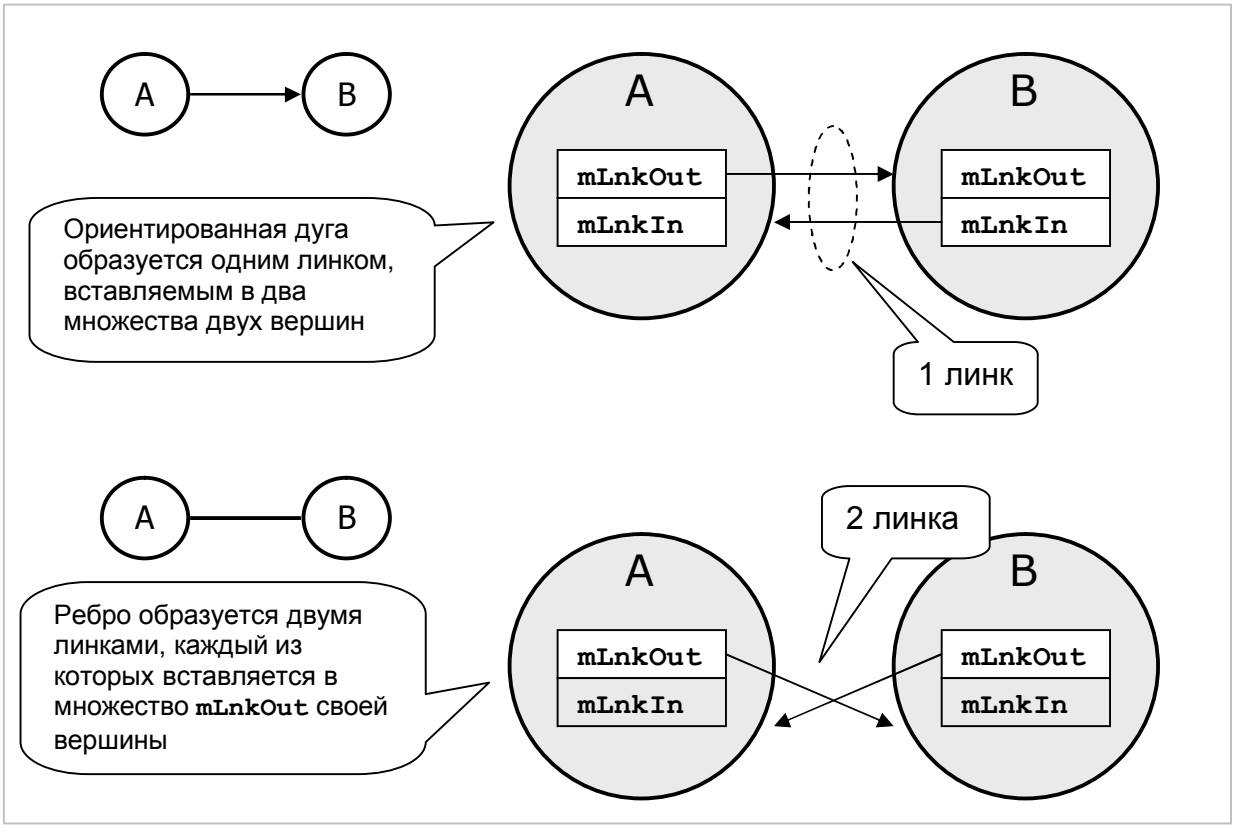


Рис. 12-4 — Организация связей в ориентированном (вверху) и неориентированном (внизу) графах

В следующей главе рассмотрим конструирование, уничтожение, ввод и вывод графов.

12.4. Итоги

12.4.1. Табличные представления графа — матрица инциденций и матрица смежности — редко применяются в программировании ввиду неэффективного использования памяти.

12.4.2. Для представления графа множествами будут использованы три класса объектов: собственно граф (**`TGraph`**), вершина графа (**`TNode`**) и связь (**`TLink`**). Все они являются наследниками класса **`TItem`**.

12.4.3. Вершины графа и связи снабжаются рядом полей, часть из которых выполняет служебные функции, а другие нужны для решения тех или иных задач на графах.

12.4.4. Направленная дуга орграфа организуется одним объектом (линком), который вставляется в два множества двух вершин: исходящей и входящей.

12.4.5. Ребро неориентированного графа организуется двумя объектами-линками, каждый из которых вставляется только в множество исходящих связей своей вершины.

12.5. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
√ 12	Хаггарти Р.	Дискретная математика для программистов	

Глава 13
Создание и ввод-вывод графов

Забегая немного вперёд, отмечу, что главной нашей заботой будет абстрактный класс **TGraph**. Он будет основой для создания графов любых типов, вершины которых могут содержать произвольную информацию: символы, числа, строки или сложные конгломераты данных (даже другие графы!). В классе **TGraph** будут накоплены методы, решающие типовые задачи на графах, причём реализация этих методов не будет зависеть от типа вершин. Тем самым ко всем наследникам базового класса перейдут решения типовых задач на графах.

Однако объекты абстрактных классов создать нельзя, реальным программам подавай реальные данные. По этой причине, наряду с базовыми классами **TGraph** и **TNode** нам следует создать их наследников, содержащих какие-либо данные — например, символы. Символы удобно отображать на экране и вводить-выводить через текстовые файлы, а для отладки и демонстрации методов графа их вполне достаточно. Классы-наследники **TGraphChars** и **TNodeChar** выделены в отдельный модуль **GrChars**.

13.1. Конструкторы базовых классов

Рассмотрим вначале конструкторы базовых классов, ниже показан конструктор графа (модуль **Graph**):

```
constructor TGraph.Create(const aName: string;
                          aDir, aLNodes, aLLinks: boolean);
begin
  inherited Create;          // вызов унаследованного конструктора
  mName:= aName;             // имя графа
  mDirect:= aDir;            // признак орграфа
  mLoadNodes:= aLNodes;      // признак нагруженных вершин
  mLoadLinks:= aLLinks;      // признак нагруженных связей
  mNodes:= CreateSet;        // пустое множество вершин
end;
```

Он выполняет лишь минимальную работу, заполняя часть полей и создавая пустое множество вершин. Наполнение пустого графа и связывание его вершин происходит в других методах:

```
// Вставка узла графа, возвращает FALSE при попытке вставить дубликат

function TGraph.InsertNode(aNode: TNode): boolean;
begin
  Result:= mNodes.Insert(aNode)
end;

// Установка связи дугой или ребром
// aSource - вершина-источник
// aDest   - вершина-приёмник
// aVal    - "вес" дуги или ребра
```

```
procedure TGraph.SetLink(aSource, aDest: TNode; aVal: integer);
begin
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  aSource.MakeLink(aDest, aVal);          // установка прямой связи
  if not mDirect                          // если не орграф,
    then aDest.MakeLink(aSource, aVal);   // то установка обратной связи
end;
```

Указанные выше методы будут вызываться из конструктора класса-наследника **TGraphChars**.

Теперь взглянем на два метода объекта-вершины **TNode**. Конструктор вершины будет вызываться, как и положено, из конструктора наследника, а метод **MakeLink** вызывается при установке связи из метода **TGraph.SetLink**.

```
constructor TNode.Create(aVal: integer; aOwner: TGraph);
begin
  inherited Create;              // унаследованный конструктор
  mOwner:= aOwner;               // владелец узла (граф)
  if mOwner.mLoadNodes           // если вершины нагружены
    then mValue:= aVal;          // то запомнить вес
  mLnkOut:= CreateSet;           // пустое множество исходящих связей
  if mOwner.mDirect              // если орграф
    then mLnkIn := CreateSet;    // то создать пустое множество входящих связей
end;

// Установка связи между данным узлом и заданным
// aDest - вершина-приёмник
// aVal  - "вес" связи

procedure TNode.MakeLink(aDest: TNode; aVal: integer);
var L: TLink;
begin
  if not Assigned(aDest) then Exit;
  L:= TLink.Create(Self, aDest, aVal, mOwner.mLoadLinks);
  // попытка вставить исходящую связь
  if not mLnkOut.Insert(L) then begin
    L.Free;  // если уже установлена, то удаляем ненужный линк
    Exit;
  end;
  if mOwner.mDirect                 // если орграф
    then aDest.mLnkIn.Insert(L); // то вставляем в приёмнике как входящую
end;
```

Наконец, конструктор объекта-связи выглядит так:

```
constructor TLink.Create(aOwner, aDest: TNode; aVal: integer;
                         aLoadLinks: boolean);
begin
  inherited Create;           // унаследованный конструктор
  mOwner:= aOwner;            // узел-источник связи и её владелец
  mDest:= aDest;              // узел-приёмник связи
  mLoadLinks:= aLoadLinks;    // признак нагруженной связи
  if aLoadLinks               // если нагружена
    then mValue:= aVal        // запоминаем "вес"
    else mValue:= 1;          // а по умолчанию = 1
end;
```

13.2. Наследники

Носителем информации в графе-наследнике будет объект-вершина **TNodeChar**. Отметим, что его класс-предок **TNode** снабжен лишь служебными полями (если не считать веса **mValue**), поэтому в вершине-наследнике добавлено поле **mName**, хранящее один символ (см. модуль **GrChars**):

```
TNodeChar = class (TNode)
    private
      function GetLinkByName(aName : char): TLink;
    public
      mName  : char;  // хранимый символ
      constructor Create(aName: char; aVal: integer; aOwner: TGraph);
      function  Compare(arg: TItem): TCompare; override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;
```

Описание объекта-наследника графа **TGraphChars** выглядит так:

```
TGraphChars = class (TGraph)
    private
      function  GetLinkByName(aSource, aDest : char): TLink;
      procedure MakeLink(aSource, aDest : char; aVal : integer);
    public
      constructor Load(const aName: String);
      constructor GenRandom(aDir: boolean;
                            aLoadNodes, aLoadLinks,
                            aNodes, aLinks : integer);
      procedure Save(const aName: String);
      function  GetNode(aName : char): TNodeChar;
  end;
```

Наряду с тремя вспомогательными методами, здесь видны два конструктора и метод **Save**, сохраняющий граф в текстовом файле. Конструктор **Load** и метод **Save** взаимно согласованы так, что первый «умеет» вводить данные из файла, созданного вторым. Описание и образец такого файла для орграфа с взвешенными вершинами представлен ниже.

13.3. Формат файла для ввода и вывода графа

Первая строка файла с графом содержит произвольный текст — это комментарий к графу. Вторая — число 0 или 1, где 0 задаёт неориентированный граф, а 1 – орграф. Третья и четвёртая строки задают аналогично признаки ненагруженных или нагруженных вершин и рёбер. В пятой строке задано количество вершин, а имена вершин перечислены в шестой строке, причём для графа с взвешенными вершинами после знака «=» указывается вес вершины.

Последующие строки содержат связи каждой из вершин. Если дуга или ребро взвешены, то после каждой связи через знак «=» указывается вес связи, например:

```
A -> B=3 E=7 F=2
```

Вот пример файла для загрузки орграфа с взвешенными вершинами:

```
Орграф с взвешенными вершинами - комментарий
1 - граф(0), орграф(1)
1 - нагруженность вершин
0 - нагруженность рёбер (дуг)
13 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8 I=9 J=10 K=11 L=12 M=13
A -> B E F
B -> A
C -> B D E
D -> I
E -> A G
F -> E H J
G -> D
H -> G J
I -> G
J -> H
K -> J L
L -> H K M
M -> K
```

Такой файл удобен для ручной правки. Ознакомиться с методами загрузки и выгрузки файла можно по листингу модуля **GrChars**.

13.4. Генерация случайного файла

Другим средством, создающим граф, является конструктор случайного графа, объявленный так:

```
constructor TGraphChars.GenRandom(
                              aDir: boolean;  // признак орграфа
                              aLoadNodes,     // нагруженность вершин
                              aLoadLinks,     // нагруженность связей
                              aNodes,         // предельное число вершин
                              aLinks:         // предельное число связей (%)
                              integer);
```

Здесь задаются основные параметры графа, а также предельное количество вершин и связей. Количество связей задаётся в процентах от теоретического максимума (**N*N-N**). При положительных значениях чисел вершин и рёбер фактическое количество их формируется случайно, а для отрицательных — точно. Метод предназначен для тестирования программ.

13.5. Визуализация файла (Expo)

При отладке задач на графах полезно наблюдать граф в удобной для восприятия форме (или выводить в этой форме в текстовый файл). Напомню, что выполняющие эту работу методы **Print** (виртуальный) и **Expo** (статический) существуют во всех элементах, поскольку встроены в предка **TItem**.

Визуализация графа реализована в базовом классе (**TGraph.Print**), что избавляет нас от повторения этой работы в наследниках. Метод очень прост, поскольку применяет уже отработанную визуализацию множеств. Особенности наследника учтены через виртуальный метод **GetName** класса **TNode** где он объявлен абстрактным. А в классе-наследнике **TNodeChar** он реализован так:

```
function TNodeChar.GetName: string;
begin
  Result:= mName;  // возвращаем символ - название вершины
end;
```

13.6. Испытание

Для демонстрации методов конструирования и отображения графа можно воспользоваться приведенной ниже программой.

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

Const  CFile = 'Output.txt';
var Gr : TGraphChars;
    S : string;
    F : Text;
begin
  repeat
    Gr:= TGraphChars.GenRandom(   // создание случайного графа
              Boolean(Random(2)), // признак орграфа
              Random(2),          // признак нагруженных вершин
              Random(2),          // признак нагруженных дуг (рёбер)
              20,                 // вершин не более 20
              40{%}               // связей не более 40%
              );
    Gr.Expo;  // отображение на экране
    Write('Save? [Y] :'); Readln(S);
    if (S='y') or (S='Y') then begin
      // Сохранить в файле вместе с визуальным отображением
      Gr.Save(CFile);
      Assign(F, CFile);
      Append(F);
      Writeln(F,'============================');
      Gr.Print(F);
      Close(F);
      // Прочитать из файла и вывести на экран:
      Gr.Free;
      Gr:= TGraphChars.Load(CFile);
      Gr.Expo;
      Readln;
    end;
    Gr.Free;
  until S<>'';
end.
```

Результаты работы этой программы представлены ниже.

Файл, описывающий не взвешенный орграф:

```
Random: T : 0 : 0 : 14 : 20
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
14 - количество вершин
A B C D E F G H I J K L M N
A -> I J
B -> C I
C -> M
D ->
E -> G J
F ->
G ->
H -> I
I ->
J -> E H I N
K -> E G J N
L -> D
M -> J K L
N ->
```

Отображение этого орграфа на экране таково:

```
Random: T : 0 : 0 : 14 : 20
{
A  ->   { I J } : 2
B  ->   { C I } : 2
C  ->   { M } : 1
D  ->   { } : 0
E  ->   { G J } : 2
F  ->   { } : 0
G  ->   { } : 0
H  ->   { I } : 1
I  ->   { } : 0
J  ->   { E H I N } : 4
K  ->   { E G J N } : 4
L  ->   { D } : 1
M  ->   { J K L } : 3
N  ->   { } : 0
} : 14
```

Файл, описывающий граф с взвешенными вершинами и рёбрами:

```
Random: F : 1 : 1 : 13 : 10
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
13 - количество вершин
A=1 B=1 C=1 D=1 E=2 F=2 G=2 H=1 I=2 J=1 K=1 L=1 M=1
A -> B=2
B -> A=2 M=1
C -> I=2 M=1
D ->
E ->
F -> G=1
G -> F=1
H ->
I -> C=2
J ->
K ->
L ->
M -> B=1 C=1
```

Тот же файл, отображённый на экране:

```
Random: F : 1 : 1 : 13 : 10
{
A =  1 ->   { B= 2 } : 1
B =  1 ->   { A= 2 M= 1 } : 2
C =  1 ->   { I= 2 M= 1 } : 2
D =  1 ->   { } : 0
E =  2 ->   { } : 0
F =  2 ->   { G= 1 } : 1
G =  2 ->   { F= 1 } : 1
H =  1 ->   { } : 0
I =  2 ->   { C= 2 } : 1
J =  1 ->   { } : 0
K =  1 ->   { } : 0
L =  1 ->   { } : 0
M =  1 ->   { B= 1 C= 1 } : 2
} : 13
```

Обратите внимание, что вызовом **`Gr.Save('')`** на экране можно отобразить исходный файл с описанием графа.

13.7. Итоги

13.7.1. Основой для реализации графов любых типов является класс **`TGraph`**, в нём содержатся все статические методы, решающие задачи на графах. Однако этот класс не привязан к конкретным данным. Для наполнения графа данными надо создавать наследников узла и графа.

13.7.2. Созданные нами наследники **`TGraphChars`** и **`TNodeChar`** оперируют с символами. В наследнике графа предусмотрены два конструктора: для загрузки из файла, и для случайной генерации графа. Реализован также метод сохранения графа в файле.

Глава 14
Достижимость

Отныне предметом наших размышлений будет группа островов — Банановый архипелаг (рис. 14-1), где по прихоти своей мы будем строить мосты, разрушать их, и возводить вновь. Всё это будет подчинено решению задач на графах. Эти решения мы оформим как статические методы объекта класса **`TGraph`**.

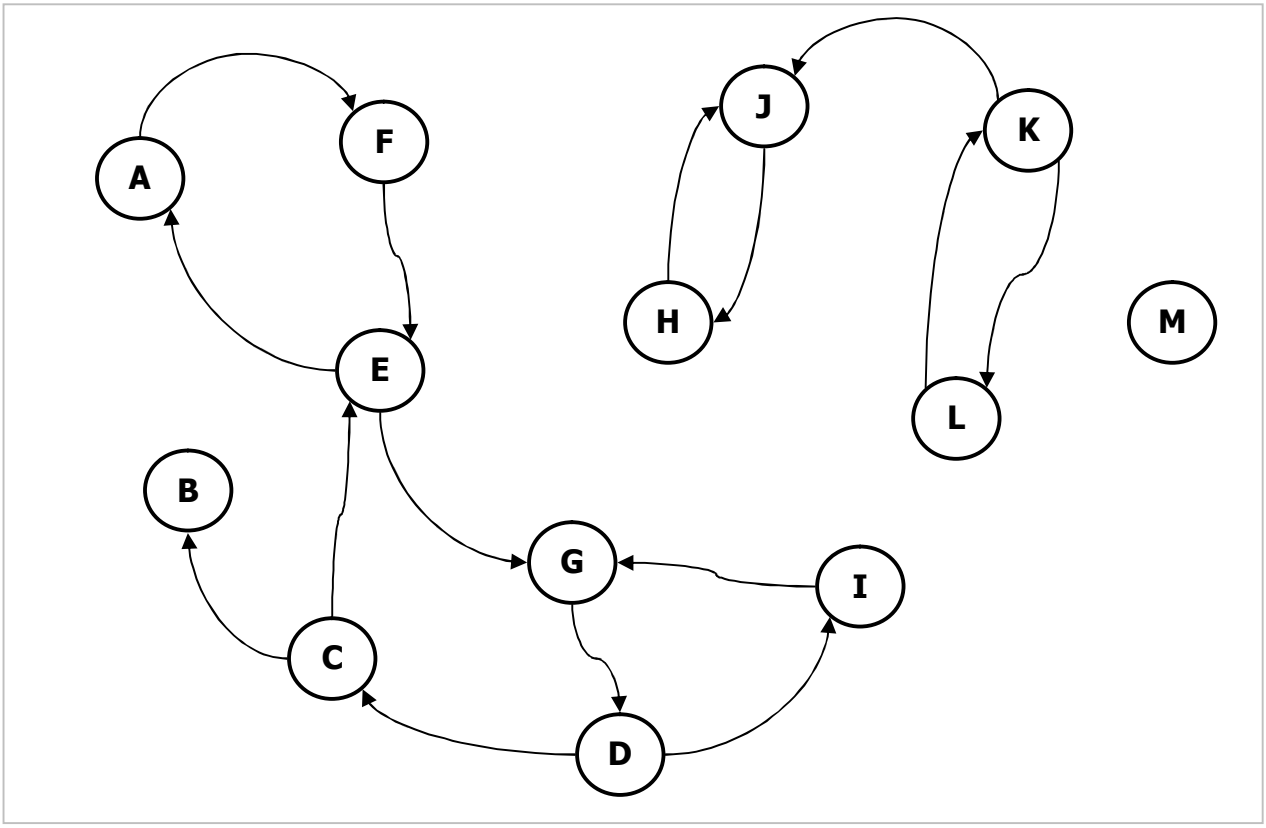


Рис. 14-1 — Мостовая сеть Банановых островов

14.1. Достижима ли вершина?

Простейшая задача о достижимости формулируется так: возможно ли, следуя по дугам или рёбрам, попасть из вершины *A* в вершину *B* ? Да или нет? Вопрос касается и графов, и орграфов. Из того, что некоторые острова или группы островов не связаны мостами, следует, что часть островов взаимно не достижима. Но даже когда все острова соединены, это не гарантирует взаимной достижимости: так будет, если часть мостов допускает движение лишь в одном направлении. К примеру, с острова *B* недостижимы другие острова (см. рис. 14-1), хотя сам он достижим из некоторых.

Для выявления достижимости построим булеву функцию — метод **`TGraph.TestLink`**, принимающую два параметра: исходную и конечную вершины. Метод будет возвращать **`TRUE`**, если возможен путь из первой вершины во вторую:

Листинг 14-1 — Проверка достижимости вершины

```
// Возвращает TRUE при наличии пути из вершины 1 в вершину 2
// (граф и орграф)

function TGraph.TestLink(arg1, arg2: TNode): boolean;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
    old : integer; // мощность предыдущей гаммы
      i : integer;
begin
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    Result:= SN1.Exist(arg2);    // конечная вершина в гамме?
    if Result then Break;        // да, выход из цикла
    old:= SN2.GetCount;          // запоминаем мощность
    // расширяем гамму SN2 := Gamma(SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    SN1.CopyItems(SN2);    // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
end;
```

Решение состоит в постепенном расширении множества смежных вершин — накоплении исходящей интегральной гаммы, начиная от исходной вершины:

$$\Gamma = \Gamma^0 + \Gamma^1 + \Gamma^2 + \ldots$$

Цикл расширения гаммы **repeat-until** прекращается либо когда целевая вершина попала в гамму Γ, либо когда расширение гаммы стало невозможным, и тогда функция возвращает **FALSE**.

Текущая гамма — множество вершин **SN2** — расширяется циклическим вызовом метода **TNode.OutGammaAdd**, который добавляет к множеству **SN2** исходящую гамму для каждого узла из **SN1**. Если после попытки расширения мощность **SN2** не увеличилась, значит, дальнейшее движение по мостам невозможно. Метод **TNode.OutGammaAdd** очень прост, но будет применяться ещё не раз, и потому покажем его здесь:

```
procedure TNode.OutGammaAdd(aRes : TSet);
var L: TLink;
begin
  L:= OutLinkFirst;             // первая исходящая связь
  while Assigned(L) do begin    // перебор связей
    aRes.Insert(L.mDest);       // вставить целевую вершину в результат
    L:= OutLinkNext;            // следующая исходящая связь
  end;
end;
```

14.2. Подсчёт мостов

Слегка изменив функцию **TestLink**, можно получить количество дуг (мостов), преодолеваемых на пути из одной вершины в другую. Если вершины не достижимы, функция **TGraph.CalcSteps** возвращает минус единицу:

Листинг 14-2 — Подсчёт количества дуг на пути из вершины в вершину

```
function TGraph.CalcSteps(arg1, arg2: TNode): integer;
var SN1 : TSet;      // предыдущее множество вершин
    SN2 : TSet;      // следующее  множество вершин (накопитель)
   node : TNode;     // очередная вершина из SN1
     ok : boolean;   // признак достижения целевой вершины
    old : integer;   // предыдущая мощность накопленной гаммы
      i : integer;
begin
  Result:= 0;
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    ok:= SN1.Exist(arg2);  // признак достижения целевой вершины
    if ok then Break;
    old:= SN2.GetCount;    // запоминаем мощность до расширения
    // Накопление гаммы:  SN2 := Gamma (SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы SN1
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    Inc(Result);
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
  if not ok then Result:= -1;   // признак недостижимости
end;
```

Следующая программа может быть использована для испытания рассмотренных методов:

Листинг 14-3 — Проверка наличия пути между двумя вершинами

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

var Gr : TGraphChars;
    NodeA, NodeB : TNode;
    S : string;

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
```

```
  Gr.Save(CName);
  AssignFile(F, CName);
  if FileExists(CName) then Append(F) else Rewrite(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - - - -');
  Writeln(F,'A -> B : ', Gr.TestLink(NodeA, NodeB));
  Writeln(F,'B -> A : ', Gr.TestLink(NodeB, NodeA));
  Writeln(F,'A => B : ', Gr.CalcSteps(NodeA, NodeB));
  Writeln(F,'B => A : ', Gr.CalcSteps(NodeB, NodeA));
  Writeln(F,'-----------------------------------');
  Close(F)
end;
////////////////////////////////////////////////////////////
begin
  repeat
    Gr:= TGraphChars.GenRandom(false, 0, 0, 20, 40);
    Gr.Expo;
    NodeA:= Gr.GetNode('A'); NodeB:= Gr.GetNode('B');
    Writeln('A -> B : ', Gr.TestLink(NodeA, NodeB));
    Writeln('B -> A : ', Gr.TestLink(NodeB, NodeA));
    Writeln('A => B : ', Gr.CalcSteps(NodeA, NodeB));
    Writeln('B => A : ', Gr.CalcSteps(NodeB, NodeA));
    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

Вот один из результатов, выданных программой:

```
Исходный неориентированный не взвешенный граф
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
13 - количество вершин
A B C D E F G H I J K L M
A -> C
B -> F G K
C -> A G
D -> L
E -> H
F -> B M
G -> B C H M
H -> E G
I ->
J -> M
K -> B L
L -> D K
M -> F G J
- - - - - - - - - - - - - - - - - - - -
A -> B : TRUE
B -> A : TRUE
A => B : 3
B => A : 3
```

14.3. Итоги

14.3.1. Одна вершина графа достижима из другой, если существует путь из второй в первую.

14.3.2. В неориентированном графе, где связи симметричны, достижимость вершин взаимна, в орграфе это не так.

14.3.3. Достижимость целевой вершины из исходной вершины, а также количество преодолеваемых дуг (рёбер) определяется посредством расширения множества смежных вершин (исходящей интегральной гаммы).

14.4. Задания

14.5. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
√ 6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
√ 12	Хаггарти Р.	Дискретная математика для программистов	

Глава 15
Кратчайшие пути

В отношение достижимости вершин можно задаться следующими вопросами: которые из вершин лежат на пути между исходной и конечной вершинами? И, если путей несколько, то который из них короче?

15.1. Кратчайший путь

Сначала рассмотрим решение задачи для единичных дуг или рёбер (иначе говоря, без учёта длины). Пусть требуется найти кратчайший путь из вершины *A* к вершине *G* (рис. 15-1). Сначала поставим промежуточную цель: построим вспомогательное дерево обратных связей. Для этого, начиная с исходной вершины *A* , станем обрабатывать соседей, изменяя в них два поля:

- **`mColor`** — цвет вершины, это числа: 0 - белый, 1 - серый, 2 - чёрный;
- **`mPred`** — указатель обратной связи на предшествующую вершину.

Предстоит своеобразное расширение гаммы, но с привлечением подсобного средства — очереди вершин. Вначале все вершины сбросим в исходное состояние: цвет белый, а обратный указатель пустой. Затем исходную вершину покрасим серым и поставим в очередь обработки (серость вершины — это признак пребывания её в очереди). На этом завершается начальная подготовка (рис. 15-1).

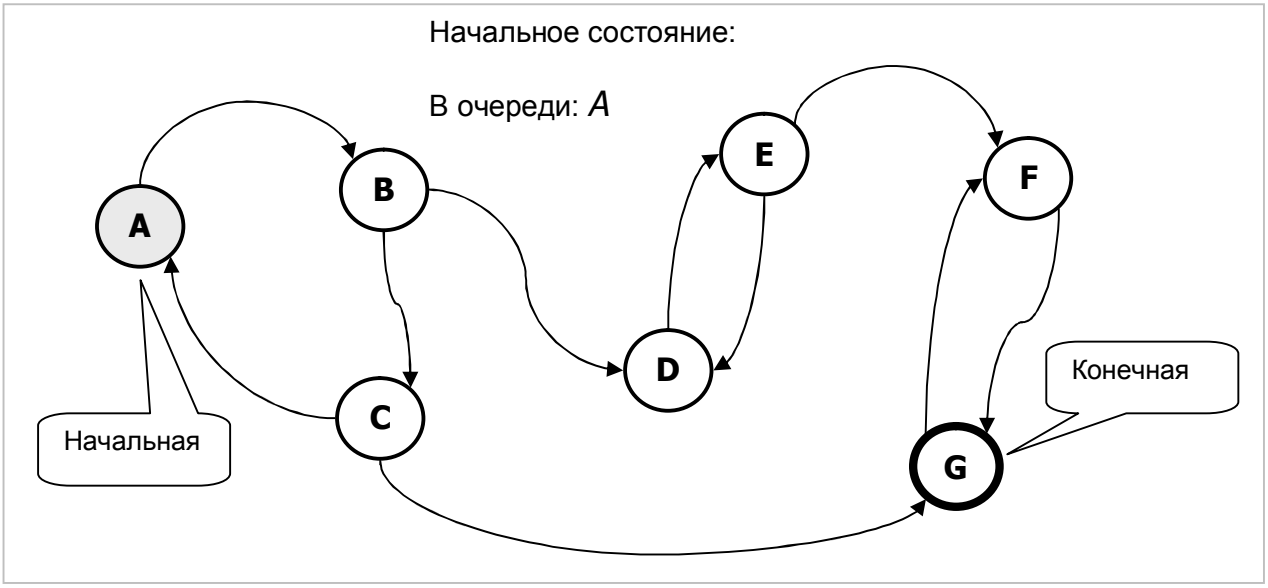


Рис. 15-1 — Начальное состояние процесса построения дерева

Далее запускаем цикл построения дерева обратных связей. Выбрав из очереди первую вершину, обрабатываем её белых соседей так: указатель обратной связи устанавливаем на текущую вершину, красим эту соседнюю вершину серым и ставим в очередь. Так продолжаем, пока соседней не окажется конечная вершина *G*, либо не будет исчерпана очередь вершин.

Два промежуточных состояния процесса показаны на следующих рисунках, где обратные связи изображены пунктиром.

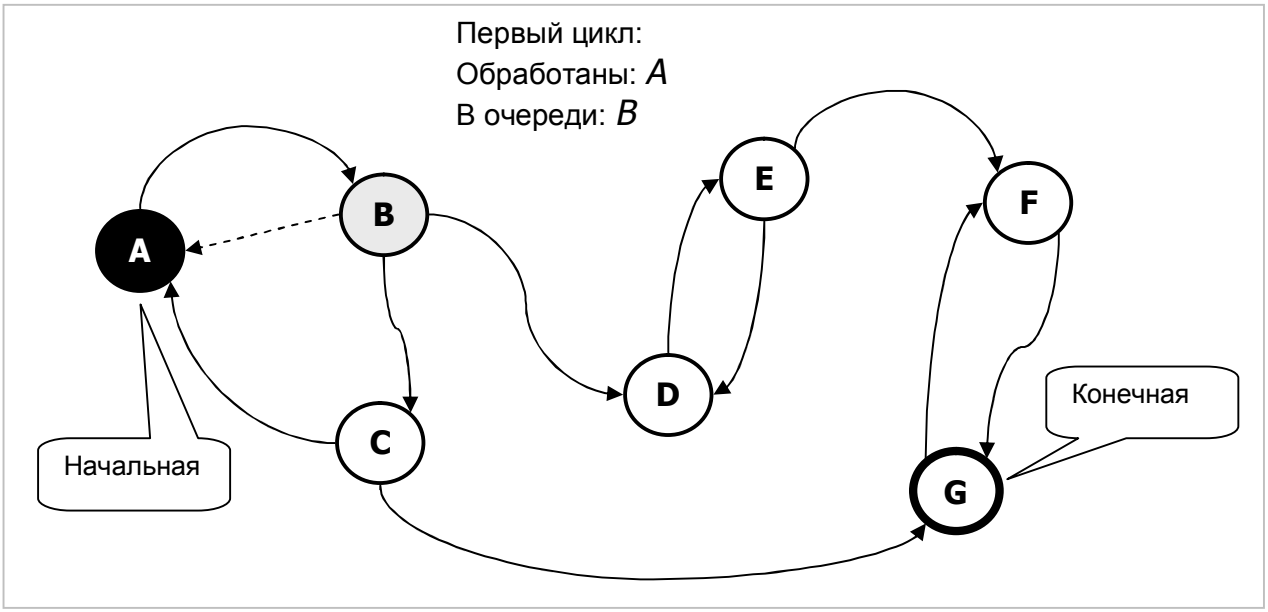


Рис. 15-2 — Состояние процесса после обработки A

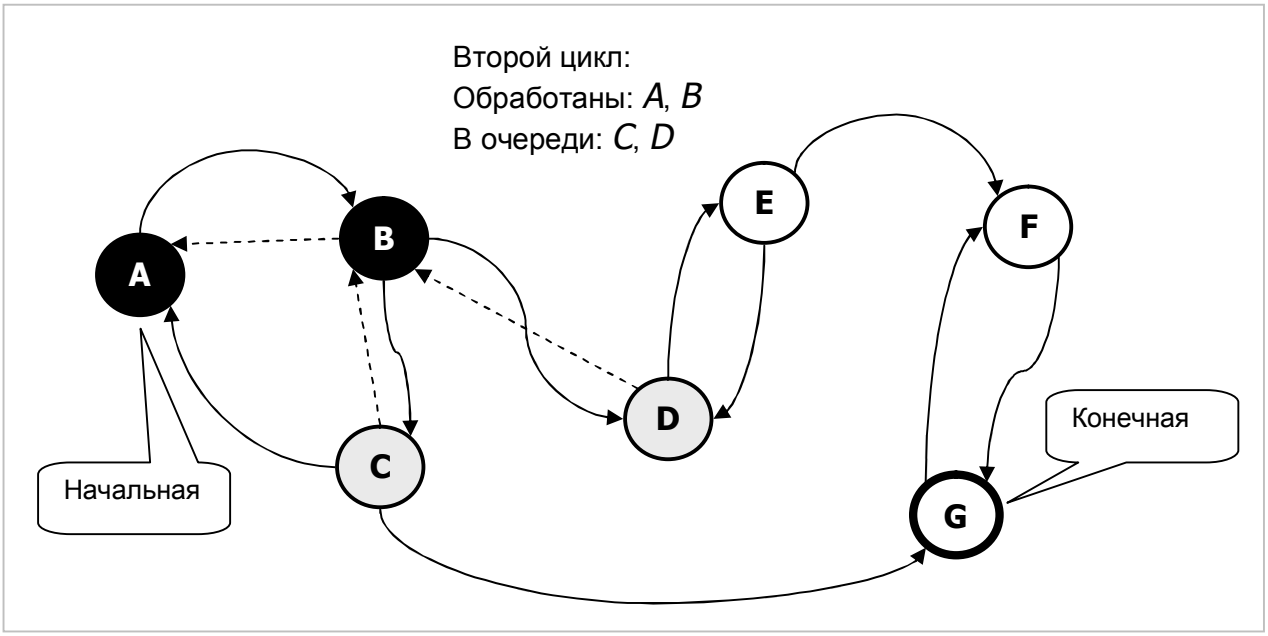


Рис. 15-3 — Состояние процесса после обработки A и B

В конце концов, когда при обработке соседей будет обнаружена конечная вершина, обратные связи будут такими, как показано на рис. 15-4. С этого момента запускается финальная фаза: следуя от конечной вершины по обратным связям строим путь от конечной вершины к начальной. Затем для получения пути от начальной вершины к конечной этот путь реверсируем. В неориентированном графе реверс не потребуется, если расширение гаммы начать с конечной вершины.

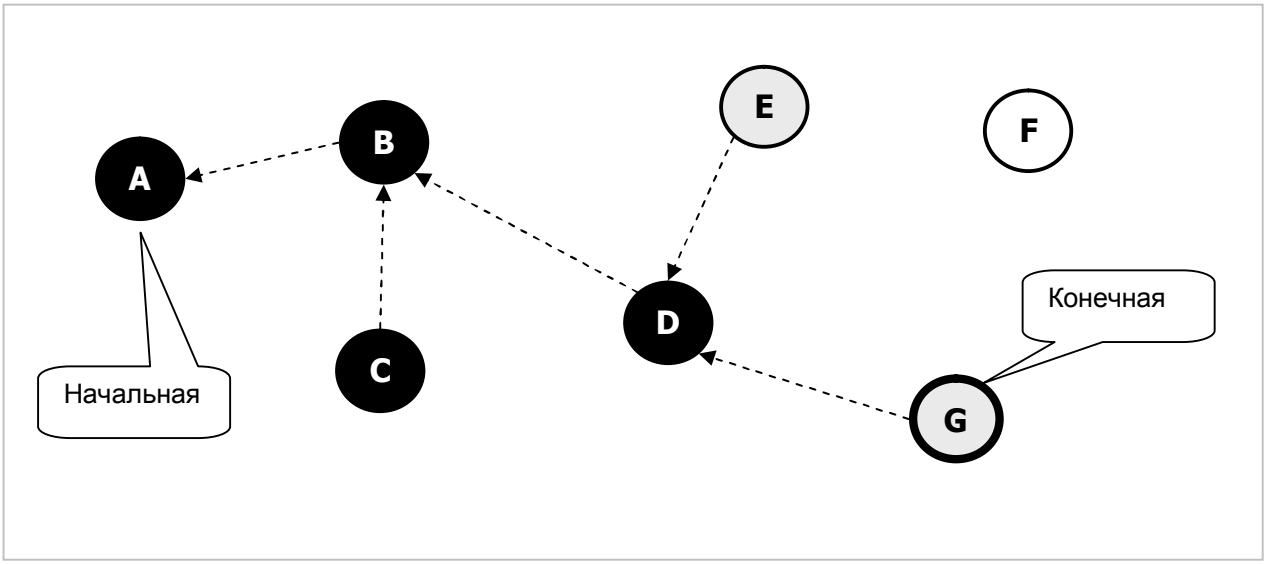


Рис. 15-4 — Дерево обратных связей при достижении конечной вершины G

Этот алгоритм реализован в методе **`TGraph.GenPath`**.

Листинг 15-1 — Построение кратчайшего пути в не взвешенном графе

```
function TGraph.GenPath(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
    Node : TNode;      // текущая вершина
    Link : TLink;      // исходящая связь
      OK : boolean;    // признак достижения целевой вершины

begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)

  ResetNodes;              // сброс узлов в исходное состояние
  aSource.mColor:= CGray;  // исходная = серая
  Que.Push(aSource);       // исходную в очередь

  repeat // повторяем, пока очередь не пуста
    OK:= false;                    // признак достижения конечной вершины
    Node:= TNode(Que.Get);         // извлечь очередную вершину
    Link:= Node.OutLinkFirst;      // первая исходящая связь
    while Assigned(Link) do begin // перебор исходящих связей
      with Link do if mDest.mColor = CWhite then begin
        // если смежная не тронута (белая)
        mDest.mPred:= Node;        // указатель на предыдущую
        OK:= mDest = aDest;        // целевая?
        if OK then Break;          // выход из цикла while
        mDest.mColor:= CGray;      // иначе красим серым
        Que.Push(mDest);           // и ставим в очередь
      end;
      Link:= Node.OutLinkNext; // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;      // обработанную вершину красим чёрным
    // выход, если достигнута целевая либо исчерпана очередь
  until OK or (Que.GetCount = 0);
```

```
  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Push(aDest);                  // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Push(Node);                 // вершину --> в результат
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;       // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

Следующая программа испытывает метод **GenPath**.

Листинг 15-2 — Программа для испытания метода GenPath

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';

var Gr : TGraphChars;
    P : TBuffer;
    NodeA, NodeB : TNode;
    S : string;

////////////////////////////////////////////////////////////
///  Сохранение избранных графов

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
  Gr.Save(CName);
  AssignFile(F, CName);
  Append(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - -');

  P:= Gr.GenPath(NodeA, NodeB);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;

  Writeln(F,'- - - - - - - - - - - - - - - - - -');

  P:= Gr.GenPath(NodeB, NodeA);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;

  Writeln(F,'- - - - - - - - - - - - - - - - - -');
  Close(F)
end;
```

```
begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 5, 5, 20, 50);
    Gr.Expo;

    NodeA:= Gr.GetNode('A');
    NodeB:= Gr.GetNode('B');
    Writeln('- - - - - - - - - - - - - - - - - -');

    P:= Gr.GenPath(NodeA, NodeB);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');

    P:= Gr.GenPath(NodeB, NodeA);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - -');

    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

15.2. Алгоритм Дейкстры

Теперь рассмотрим поиск кратчайшего пути для графов с взвешенными дугами и (или) вершинами. Пусть некий турист намерен отправиться из города *A* в город *C*. География даёт ему три маршрута, и по неясным причинам стоимость проезда по дорогам оказалась такова, как показано на рис. 15-5.

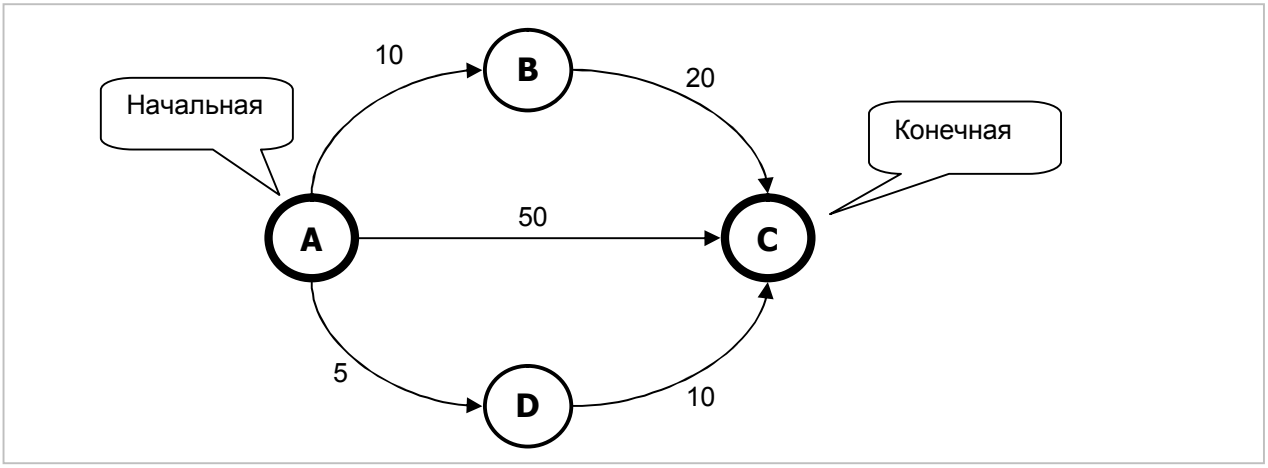


Рис. 15-5 — Возможные пути из A в C

Здесь самым дорогим является прямой путь из *A* в *C* (50 пиастров), а самым дешёвым — путь через город *D* (15 пиастров). Надо построить алгоритм, отыскивающий этот наиболее дешёвый путь (или один из путей, если существуют несколько равноценных). Вдобавок к стоимости дуг, мы учтём и стоимость промежуточных вершин, — тем самым можно моделировать затраты на ночлег в гостиницах (в нашем примере вершины не взвешены и стоимость ночлега нулевая).

В основе решения лежат те же идеи, что в первой задаче, но ссылку на предшествующую вершину будем поправлять с учётом накопленной стоимости

пути — по сумме стоимости дуг и промежуточных вершин. Пусть после обработки и почернения исходной вершины *A* в очереди оказались серые вершины *B*, *C* и *D* (рис. 15-6).

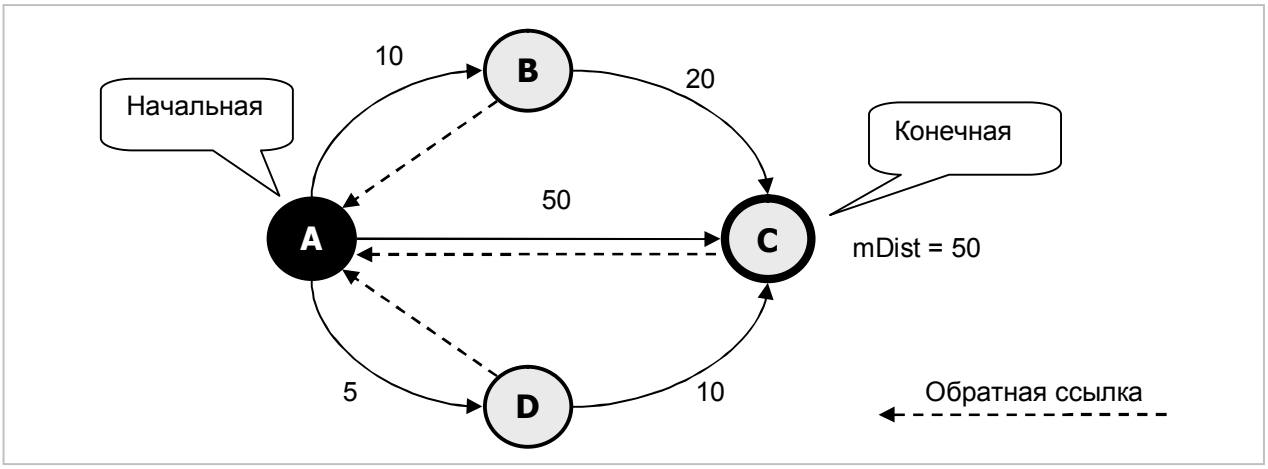


Рис. 15-6 — Состояние после почернения вершины A

Здесь пунктиром показаны обратные ссылки на предшествующие вершины. Значение поля **`mDist`** для вершины *C* составит 50 — это стоимость дуги *A-C*.

Продолжим выбирать вершины из очереди и обрабатывать их (в алфавитном порядке). При обработке вершины *B* исследуем расстояния от неё ко всем соседним вершинам. Тут выясняем, что сумма стоимостей дуг 10+20 меньше, чем отмеченная в вершине *C* стоимость 50, потому ссылку на предшествующую вершину перестраиваем в ней с *A* на *B*, а **`mDist`** исправляем на 30 (рис. 15-7).

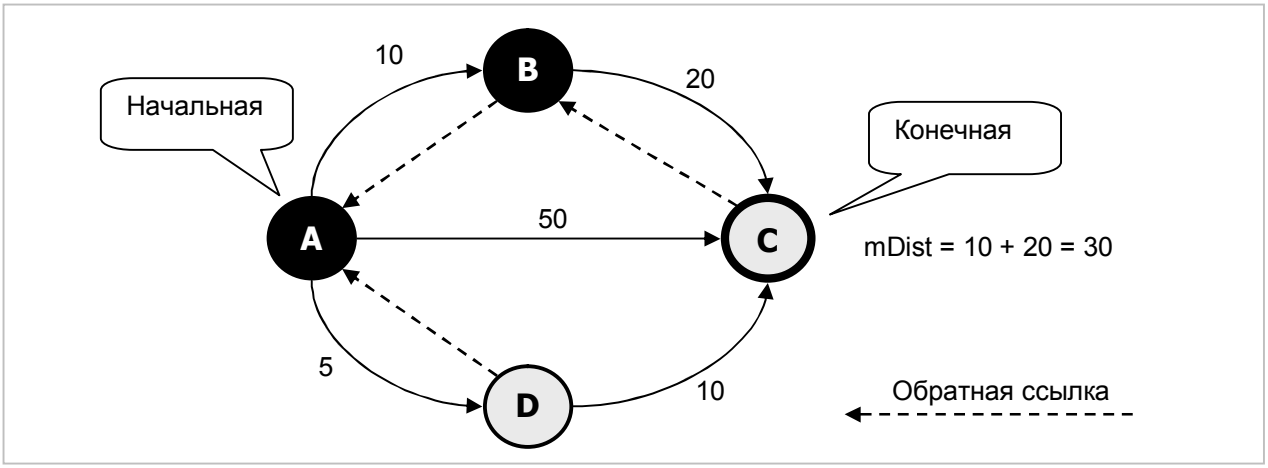


Рис. 15-7 — Состояние после обработки вершины B

При обработке вершины *C* ничего не изменится, поскольку у неё нет смежных исходящих вершин (рис. 15-8).

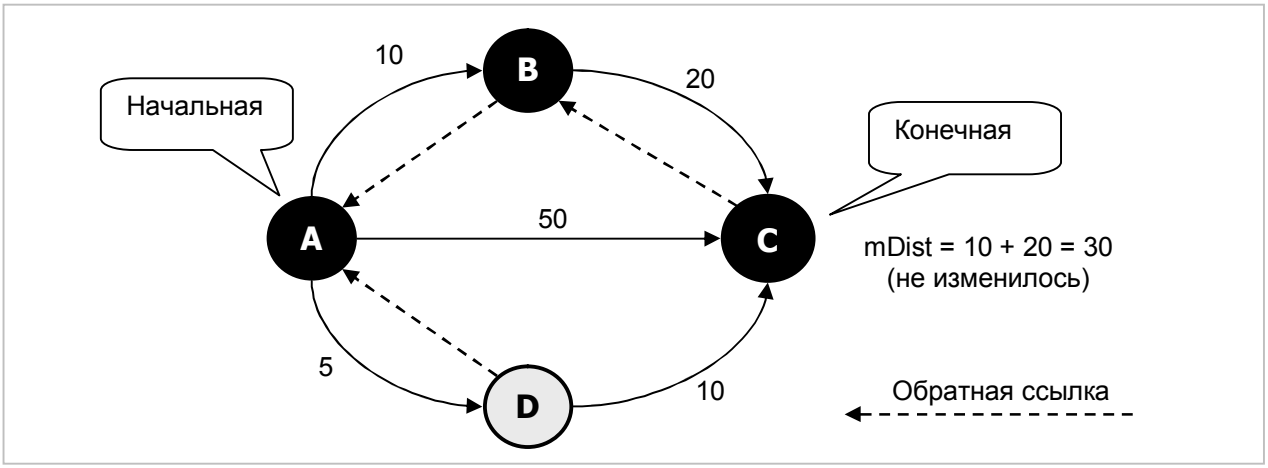


Рис. 15-8 — Состояние после обработки вершины C не изменилось

При обработке вершины *D* вновь оказывается, что смежная с нею вершина *C* может быть достигнута за меньшую цену, поэтому обратную ссылку в *C* и поле **`mDist`** вновь корректируем (рис. 15-9).

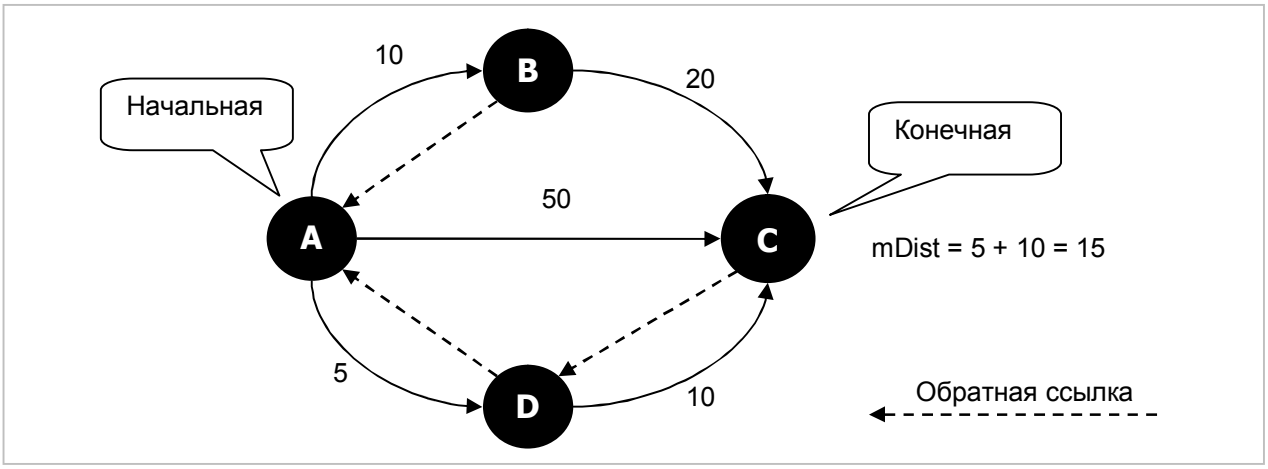


Рис. 15-9 — Состояние после обработки вершины D

Итак, наилучшим путём из *A* в *C* оказался путь через вершину *D*. Детали алгоритма показаны в следующем листинге. Отметим, что здесь нельзя прерывать цикл при первом достижении конечной вершины. Кратчайший путь непредсказуем, и потому надо обработать все доступные вершины графа (вплоть до опустошения очереди).

Листинг 15-3 — Алгоритм Дейкстры

```
function TGraph.GenDijkstra(aSource, aDest: TNode): TBuffer;
var  Que : TBuffer;  // очередь вершин
     Node : TNode;       // текущая вершина
     Link : TLink;       // исходящая связь
begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)
  ResetNodes;               // сброс узлов в исходное состояние
  aSource.mColor:= CGray;   // исходная = серая
  aSource.mDist:= 0;        // расстояние к исходной = 0
  Que.Push(aSource);        // исходную в очередь
```

```
  repeat                      // повторяем, пока очередь не пуста
    Node:= TNode(Que.Get); // извлечь очередную вершину
    Link:= Node.OutLinkFirst; // первая исходящая связь
    while Assigned(Link) do with Link do begin  // обраб. текущего линка
      if mDest.mDist > Node.mDist + Node.mValue + mValue then begin
        // поскольку расстояние в смежной больше нового,
        // обновляем расстояние и обратную ссылку
        mDest.mDist:= Node.mDist + Node.mValue + mValue;
        mDest.mPred:= Node;
        // Если смежная не стоит в очереди, то красим её белым,
        // чтобы поставить в очередь (возможно, что повторно)
        if mDest.mColor <> CGray then mDest.mColor:= CWhite;
      end;
      if mDest.mColor = CWhite then begin
        // если смежная не стояла в очереди
        mDest.mColor:= CGray;    // то красим серым
        Que.Push(mDest);         // и ставим в очередь
      end;
      Link:= Node.OutLinkNext;  // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;    // признак, что вершина обработана
  until Que.GetCount = 0;    // выход, если исчерпана очередь

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая вершина
    Que.Clear;                        // очистить буфер
    Que.Push(aDest);                  // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Push(Node);                 // вершину --> в результат
      if Node = aSource then Break;   // выход, если достигли исходную
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;   // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

Для проверки метода служит следующая программа.

Листинг 15-4 — Программа для проверки алгоритма Дейкстры

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  GrChars in '..\Common\GrChars.pas';


var Gr : TGraphChars;
    P : TBuffer;
    NodeA, NodeB : TNode;
    S : string;
```

```
//  Сохранение путей в файле

procedure Save;
const CName = 'Out.txt';
var F: TextFile;
begin
  Gr.Save(CName);
  AssignFile(F, CName);
  Append(F);
  Writeln(F,'- - - - - - - - - - - - - - - - - - -');
  P:= Gr. GenDijkstra (NodeA, NodeB);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;
  Writeln(F,'- - - - - - - - - - - - - - - - - - -');
  P:= Gr. GenDijkstra (NodeB, NodeA);
  if Assigned(P) then begin P.Print(F);  P.Free;  end;
  Writeln(F,'- - - - - - - - - - - - - - - - - - -');
  Close(F)
end;
////////////////////////////////////////////////////////////
begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 5, 5, 20, 50);
    Gr.Expo;

    NodeA:= Gr.GetNode('A'); // взять ссылку на A
    NodeB:= Gr.GetNode('B'); // взять ссылку на B
    Writeln('- - - - - - - - - - - - - - - - - - -');
    P:= Gr. GenDijkstra (NodeA, NodeB);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    P:= Gr. GenDijkstra (NodeB, NodeA);
    if Assigned(P) then begin P.Expo;  P.Free;   end;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    Write('s - Save, q - Quit : '); Readln(S);
    if S='s' then Save;
    Gr.Free;
  until S='q';
end.
```

15.3. Навигационная карта

15.3.1. Основная идея

Любому, кто плутал в незнакомой местности, знакома радость от находки указателя на разыскиваемый объект: следуя по цепочке таких промежуточных указателей, легко отыскать дорогу.

Сходную проблему решают в компьютерных сетях. Отдельные узлы этих сетей могут быть источниками информации, её получателями, а также промежуточными узлами: маршрутизаторами или роутерами (от англ. *Router*). Путь сетевых пакетов от источника к получателю обычно пролегает через несколько промежуточных узлов. Получив пакет, узел решает, которому из соседей передать его с тем, чтобы пакет достиг конечного адресата по кратчайшему пути. При этом роутер не «беспокоит Дейкстру» по поводу каждого пакета, а пользуется заранее подготовленной «картой» — набором готовых указателей.

Организуем в графе нечто подобное дорожным указателям с тем, чтобы составлять кратчайшие маршруты между любыми парами вершин по возможности быстро и просто.

Первое что приходит в голову: найти маршруты между всеми парами вершин и хранить их в готовом виде. Избыточность такого решения в смысле занимаемой памяти очевидна. А время? Если в графе имеется **N** вершин, то количество путей между всеми парами вершин составит **N• (N-1)**. Поиск каждого из них методом Дейкстры требует обработки **N** вершин, стало быть, предварительные затраты времени будут пропорциональны кубу от размера графа (**N^3**).

С другой стороны, ясно, что путнику, следующему, к примеру, из города *E* (рис. 15-10) в этой начальной точке не обязательно знать весь маршрут к цели, — надо выбирать лишь один из немногих исходящих из *E* путей. Следуя, к примеру, из *E* в *C*, он найдёт подходящий указатель, и двинется сначала в город *B*, где его ждёт следующий указатель на город *C*.

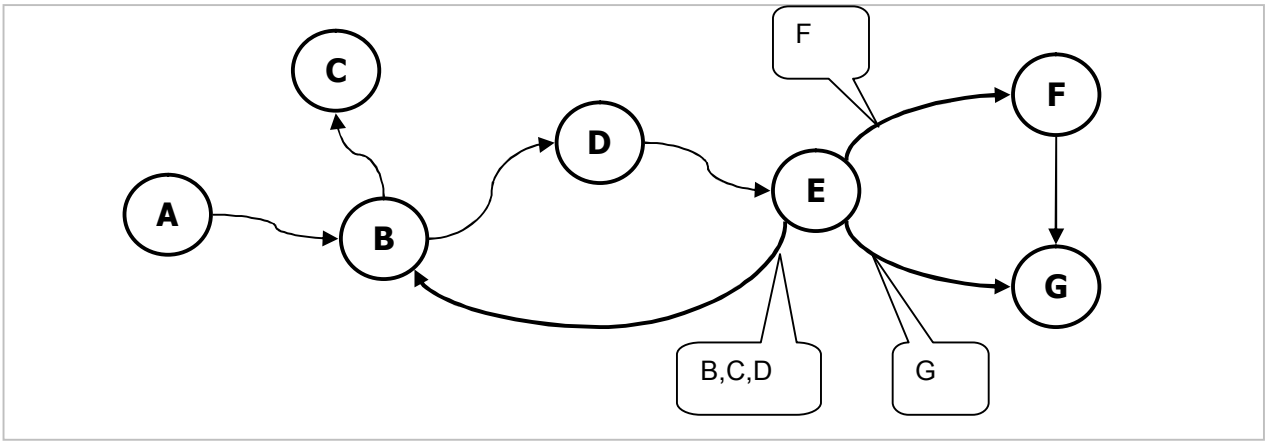


Рис. 15-10 — Дальние указатели кратчайших путей из вершины E

Принятое здесь название указателей — дальние — весьма условно, поскольку такой указатель ссылается лишь на ближайшую вершину, лежащую на искомом пути. Структура этого указателя такова:

```
TFarLink = class(TItem)      // Дальний указатель
    mNodeFar : TNode;        // целевая вершина
    mNodeNear: TNode;        // ближайшая вершина на пути к целевой
    mDist    : integer;      // расстояние от текущей к целевой
    mStep    : integer;      // номер этапа, используется при построении карты
    constructor Create(aNear, aFar: TNode; aDist: integer);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile);  override;
  end;
```

Пусть нам удалось как-то организовать в каждой вершине графа (в поле **TNode.mFarLinks**) множество таких указателей. Тогда путь между исходной вершиной **aSource** и конечной **aDest** можно построить за линейное время, следуя по дальним указателям, что реализовано в методе **TGraph.GenQuickPathStr**:

Листинг 15-5 — Формирование кратчайшего маршрута по карте маршрутизации

```
function TGraph.GetQuickPathStr(aSource, aDest: TNode): String;
var Next : TNode;
      FL : TFarLink;
       S : String;
begin
  Result:= aSource.GetName;
  // Если карта ещё не построена, то строим её
  if not mInitMap then InitMap_Floyd;
  mInitMap:= true; // Уст. признак инициализации карты маршрутизации
  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;   // если не достижима
  // Следуем по маршруту:
  repeat
    Result:= Result + ' -> ' + Next.GetName;
    Next:= Next.GetNear(aDest);
  until Next = aDest;
  FL:= aSource.GetFarLink(aDest); // дальний указатель на конечную
  Str(FL.mDist, S);               // растояние до конечной
  Result:= Result + ' -> ' + Next.GetName + ' : ' + S;
end;
```

Здесь переход к следующей вершине выполняется методом **`TNode.GetNear`**, который ищет ближайшую вершину перебором дальних связей (см. листинг).

Обратите внимание на оператор в начале метода **`GenQuickPathStr`**:

```
  // Если карта ещё не построена, то строим её
  if not mInitMap then InitMap_Floyd;
```

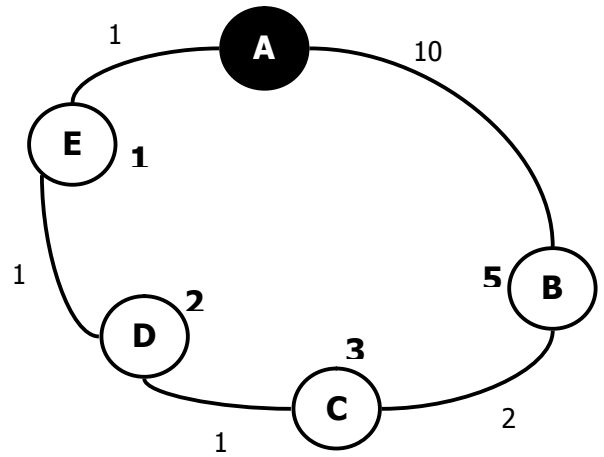
Он проверяет, проделана ли предварительная работа по организации дальних связей (флаг **`mInitMap`**), и если нет, то вызывает вспомогательный метод **`InitMap_Floyd`**, — этому методу уделим дальнейшее внимание.

15.3.2. Алгоритм Флойда-Уоршелла

Итак, для быстрого поиска кратчайших путей между двумя произвольными вершинами необходимо предварительно расставить в вершинах графа подходящие дальние указатели. Здесь можно призвать алгоритм Дейкстры, применяя его к каждой паре вершин, но метод Флойда и Уоршелла быстрее, его мы и рассмотрим.

Суть алгоритма Флойда-Уоршелла состоит в следующем. Пусть на каком то этапе решения известен путь и соответствующее ему расстояние между парой вершин, причём на этом пути лежит **`M`** промежуточных вершин (**`M=0...N-2`**). Если на следующем этапе алгоритма будет обнаружен менее длинный путь между этими же вершинами, но содержащий больше, чем **`M`**, промежуточных вершин, то прежний путь и расстояние заменяются новыми.

Рассмотрим алгоритм на примере графа, изображённого на рис. 15-11. Рядом с рёбрами указаны их длины, а рядом с вершинами — кратчайшие расстояния к вершине *A*. Мы видим, что хотя вершины *A* и *B* соединены ребром непосредственно, кратчайший путь между ними пролегает через ряд других вершин и длина его (стоимость) составляет 5.



Граф, в котором кратчайшие пути из вершины *A* ко всем остальным вершинам лежат через вершину *E*.

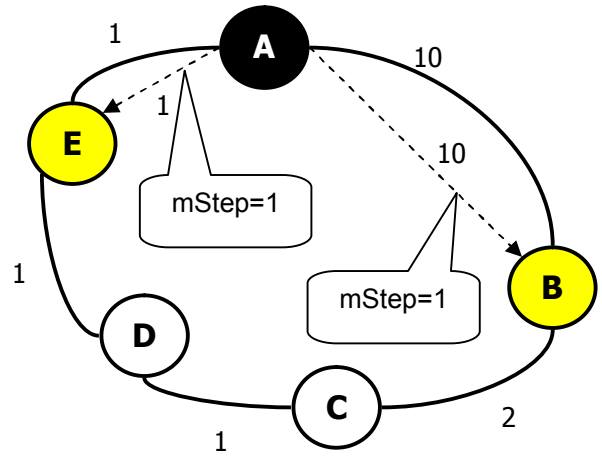
Рис. 15-11 — Пример циклического графа

Будем строить карту для этого графа по Флойду-Уоршеллу, прослеживая формирование дальних указателей из вершины *A*. Отмечу, что параллельно будут формироваться аналогичные указатели из других вершин графа, хотя для упрощения рисунка они не показаны (формирование прочих указателей можно проследить в следующих далее таблицах).

Итак, в нулевом, подготовительном цикле алгоритма, разместим в каждой вершине множество дальних указателей на все вершины графа (см. описание класса **`TFarLink`**). Один из них будет ссылаться на саму вершину *A*, два других — на ближайших соседей *B* и *E*, — на рис. 15-12 они показаны пунктиром. Дальние указатели на вершины *C* и *D* на этом этапе существуют, но ещё не определены, и на рисунке не показаны. Их поля **`mNodeNear`** будут пусты, а поля **`mDist`** равны условной бесконечности (**`MaxInt`**).

Точно так же подготавливаются дальние указатели из других вершин, все они показаны в табл. 15-1. Сформированные на этом этапе указатели ссылаются пока лишь на соседние вершины, их поля **`mStep`** установлены в единицу, а это значит, что они будут обработаны в первом цикле.

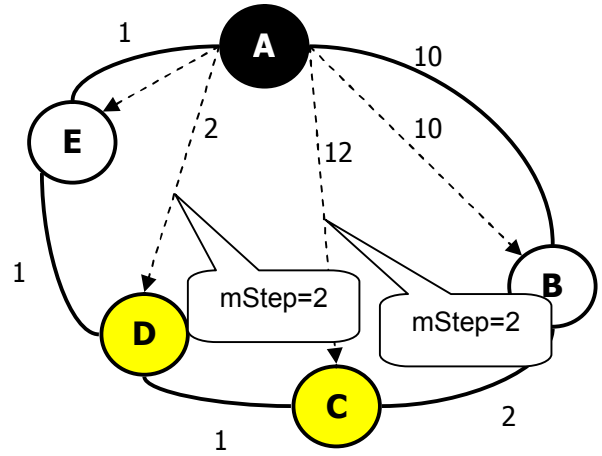
После начальной подготовки выполняем циклы обработки вершин. В первом цикле в каждой вершине обрабатываем дальние указатели, помеченные числом 1 (поле **`mStep=1`**). Через эти указатели становятся доступными вершины, отстоящие через одно ребро (соседи соседей). Если текущая длина дальнего указателя к новой вершине окажется больше, чем сумма расстояний от исходной к промежуточной и от промежуточной к конечной, то дальний указатель корректируется: изменяется расстояние и ссылка на ближайшую вершину.



Исходное состояние дальних указателей из вершины *A*.
Определены лишь два указателя на соседние вершины.
В следующем (первом) цикле будут обработаны *B* и *E*.

Рис. 15-12 — Исходное состояние перед первым циклом алгоритма

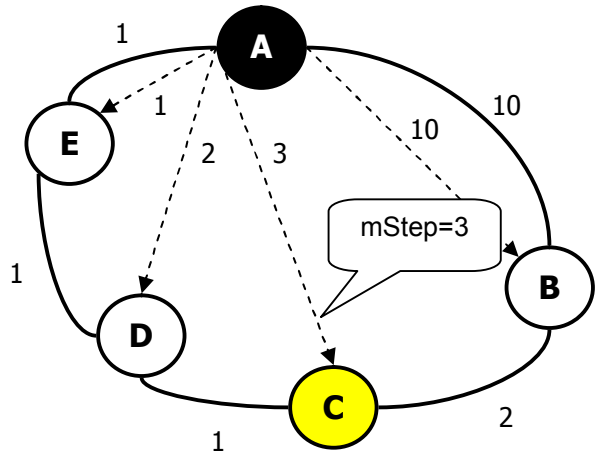
На рис. 15-13 показаны дальние указатели вершины *A* после первого цикла. Здесь из вершины *B* по цепочке *A→B→C* стала доступна вершина *C* на расстоянии 10+2=12. Из вершины *E* по цепочке *A→E→D* стала доступна вершина *D* на расстоянии 1+1=2. Обновлённые указатели к вершинам *C* и *D* отмечены числом 2, стало быть, будут обработаны на втором цикле.



Состояние дальних указателей из вершины *A* после 1-го цикла.
Определены ещё два указателя на вершины *C* и *D*.
Указатели к вершинам *B* и *C* не оптимальны (расстояния 10 и 12 соответственно).
Во втором цикле будут обработаны *C* и *D*.

Рис. 15-13 — Состояние после 1-го цикла

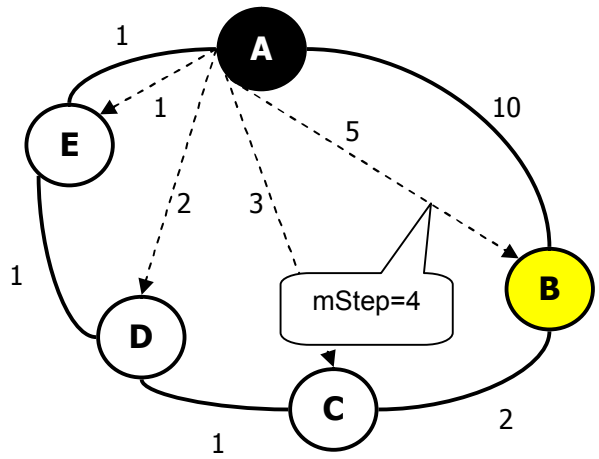
На втором цикле из вершины *C* по цепочке *A→B→C→D* делается доступной вершина *D*, но этот второй путь длиннее существующего (12+1), и потому дальний указатель на *D* не изменяется. Одновременно вершина *C* делается доступной из *D* по менее длинному пути *A→E→D→C* (2+1), потому указатель *A→C* перестраивается на путь через вершину *E* , что показано на рис. 15-14. Этот указатель помечается числом 3, чтобы быть обработанным в третьем цикле.



Состояние дальних указателей из вершины *A* после 2-го цикла. Указатель на вершину *C* изменился так, что путь к ней пролёг через вершину *E*. а расстояние уменьшилось до 3. В третьем цикле будет обработана вершина *C*.

Рис. 15-14 — Состояние после 2-го цикла

В третьем цикле обрабатываем только вершину *C*. Здесь вершина *B* стала доступна через неё по цепочке *A→E→D→C→B* с расстоянием 3+2, что меньше 10. Потому указатель на неё перестраиваем: теперь путь к *B* лежит через *E*, а расстояние составит 5 (рис. 15-15).



Состояние дальних указателей из вершины *A*. после 3-го цикла. Указатель на вершину *B* изменился так, что путь к ней стал пролегать через вершину *E*. а длина уменьшилась до 5.

Рис. 15-15 — Состояние после 3-го цикла

Сколько циклов потребуется на формирование всех дальних указателей? Из того, что длиннейший путь между вершинами содержит не более **`N-2`** промежуточных вершин, а в каждом цикле добавляется не более одной такой вершины, количество циклов составит не более **`N-2`**. Реально их может быть и меньше: когда в очередном цикле не обновится ни один дальний указатель, работу прекратим. Отметим также, что вместо пометок **`mStep`** здесь можно применить очередь дальних указателей, тогда алгоритм завершится по исчерпании этой очереди.

В следующих далее таблицах показаны этапы формирования дальних указателей для всех вершин графа. В левом столбце даны исходные вершины, в

верхней строке — конечные. В клетках показаны ближайшие вершины на пути от исходной вершины к конечной и расстояния между ними. Пустые клетки соответствуют временно не определённым дальним указателям (**`mNodeNear=nil`**). Жёлтым цветом отмечены указатели, которые будут обрабатываться на данном этапе (стоят в очереди обработки).

Табл. 15-1 — Дальние указатели перед 1-м этапом (исходное состояние)

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10			E = 1
B --> A,C	A = 10	B = 0	C = 2		
C --> B,D		B = 2	C = 0	D = 1	
D --> C,E			C = 1	D = 0	E = 1
E --> A,D	A = 1			D = 1	E = 0

Табл. 15-2 — Дальние указатели перед 2-м этапом

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10	B = 12	E = 2	E = 1
B --> A,C	A = 10	B = 0	C = 2	C = 3	A = 11
C --> B,D	B = 12	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	A = 11	D = 2	D = 1	E = 0

Табл. 15-3 — Дальние указатели перед 3-м этапом

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	B = 10	E = 3	E = 2	E = 1
B --> A,C	A = 10	B = 0	C = 2	C = 3	C = 4
C --> B,D	D = 3	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	D = 4	D = 2	D = 1	E = 0

Табл. 15-4 — Дальние указатели перед 4-м этапом (итог)

Исходные вершины	Конечные вершины				
	A	B	C	D	E
A --> B,E	A = 0	E = 5	E = 3	E = 2	E = 1
B --> A,C	C = 5	B = 0	C = 2	C = 3	C = 4
C --> B,D	D = 3	B = 2	C = 0	D = 1	D = 2
D --> C,E	E = 2	C = 3	C = 1	D = 0	E = 1
E --> A,D	A = 1	D = 4	D = 2	D = 1	E = 0

Ниже представлен листинг метода **`TGraph.InitMap_Floyd`**.

Листинг 15-6 — Построение карты дальних указателей методом Флойда-Уоршелла

```
procedure TGraph.InitMap_Floyd;
  //- - - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей
  procedure InitFarLinks(aNode: TNode);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
  begin
    PosPush;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // связь на себя
      end else with aNode do begin
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = Node then begin
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue + Ord(mLoadNodes)*Node.mValue;
            FL.mStep:= 1;      // указатель будет обработан на первом шаге
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end; // while
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;
    end;
    PosPop;
  end;
  //- - - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink; // указатель из вершины aNode
```

```
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
              then Result:= true;            // признак того, что изменён
          FL2:= GetNext as TFarLink;         // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - -
var  Node : TNode;    // текущая вершина
     Step : integer;  // этап обработки (номер цикла)
     Flag : boolean;  // признак продолжения обработки

begin { InitMap_Floyd }

  // Предварительная очистка карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin         // перебор вершин
    with Node do begin
      if Assigned(mFarLinks)            // если карта существует
        then mFarLinks.ClrAndDestroy    // очищаем
        else mFarLinks:= CreateSet;     // иначе создаём пустую
    end;
    InitFarLinks(Node);   // инициализация дальних указателей
    Node:= NodeNext;
  end;

  // Обработка вершин
  for Step:= 1 to mNodes.GetCount-2  do begin
    Flag:= false;
    Node:= NodeFirst;
    while Assigned(Node) do begin      // перебор вершин
      if Handle(Node, Step)            // если обновлялись дальние указатели
        then Flag:= true;              // то отметить
      Node:= NodeNext;
    end;
    if not Flag then Break;  // если указатели не обновлялись, то выйти
  end;
end;
```

15.3.3. Испытания

Следующую программу можно применить для испытания метода Флойда, а также для сравнения его с методом Дейкстры по скорости.

Листинг 15-7 — Испытание метода Флойда-Уоршелла

```
{$APPTYPE CONSOLE}

uses
  SysUtils, DateUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  Assembly in '..\Common\Assembly.pas',
  SetUtils in '..\Common\SetUtils.pas';

var   Gr : TGraphChars;
       S : string;
       P : TBuffer;
    Node1, Node2 : TNode;
    Start, Time : TDateTime;
    i, N1, N2 : integer;

begin
  Gr:= TGraphChars.Load('Test_1.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - - -');
  repeat
    Write('S:= '); Readln(S);     // два первых символа - имёна вершин
    if Length(S)<2 then break;    // признак завершения

    Node1:= Gr.GetNode(Upcase(S[1]));  // исходная вершина
    Node2:= Gr.GetNode(Upcase(S[2]));  // конечная вершина

    // Поиск кратчайшего пути методом Дейкстры
    P:= Gr.GenDijkstra(Node1, Node2);
    if Assigned(P) then P.Expo;
    P.Free;
    Writeln('- - - - - - - - - - - - - - - - - - -');

    // Поиск кратчайшего пути по карте,
    // построенной методом Флойда-Уоршелла
    Writeln(Gr.GetQuickPathStr(Node1, Node2));
    Writeln('- - - - - - - - - - - - - - - - - - -');

    // Сравнение быстродействия двух методов

    Start:= Now;  N1:=0;
    repeat
      for i:=1 to 1000 do begin
        P:= Gr.GenDijkstra(Node1, Node2);    // методом Дейкстры
        P.Free;
      end;
      Inc(N1);
      Time:= MilliSecondsBetween(Start, Now);
    until Time>1000;

    Start:= Now;  N2:=0;
    repeat
      for i:=1 to 1000 do begin
        P:= Gr.GenQuickPath(Node1, Node2);   // по маршрутной карте
        P.Free;
      end;
```

```
      Inc(N2);
      Time:= MilliSecondsBetween(Start, Now);
    until Time>1000;
    // Вывод соотношения быстродействий
    Writeln('QuickPath / Dijkstra = ', N2/N1:5:2);

  until false;
  Gr.Free;
end.
```

Испытания показали, что даже на небольшом графе поиск маршрутов по карте даёт многократный выигрыш времени.

15.4. Итоги

15.4.1. Кратчайший путь между двумя вершинами — это последовательность лежащих на нём вершин.

15.4.2. Кратчайший путь может определяться без учёта веса дуг и промежуточных вершин, либо с их учётом (Дейкстра).

15.4.3. При поиске кратчайших путей используют вспомогательное средство — очередь обрабатываемых вершин.

15.4.4. Для многократных вычислений кратчайших путей между произвольными парами вершин строят вспомогательную «карту» дальних указателей методом Флойда-Уоршелла.

15.5. Задания

15.6. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
√ 6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 29
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 16
Сильные связи

Разобравшись с взаимной достижимостью вершин, мы нашли способы определять сам факт достижимости, расстояние между вершинами и кратчайшие пути между ними. Теперь зададимся следующими вопросами:

- Все ли вершины графа взаимно достижимы?
- Если взаимно достижимы не все вершины, то сколько взаимно достижимых (связанных) областей содержится в графе? И как найти эти области?

Для неориентированного графа непреложен закон: если из вершины *A* достижима вершина *B*, то верно и обратное. В орграфах это возможно не всегда, и мы выясним эти закономерности.

16.1. Неориентированный граф

Вернёмся на Банановые острова, где часто гуляют тайфуны, разрушая всё на своём пути, включая мосты. Их, конечно, отстраивают, но не всегда успевают закончить работу. Вот карта островов на момент, когда лишь часть мостов восстановлена, причём эти мосты допускают двустороннее движение (рис. 16-1).

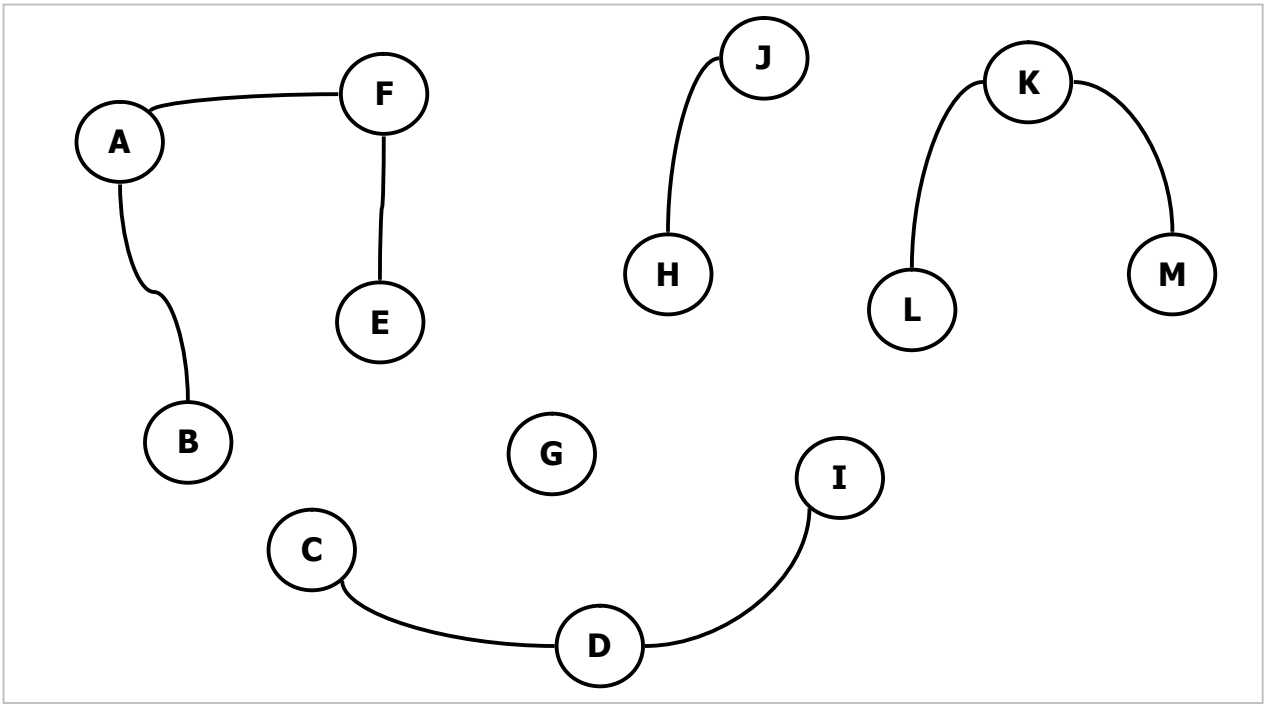


Рис. 16-1 — Соединение островов мостами с двусторонним движением

Легко догадаться, что жители островов *A-B-E-F* могут достичь друг друга по мостам — это *связная компонента* графа. Другие связные компоненты образуют подмножества вершин *C-D-I*, *G*, *H-J*, *K-L-M*, всего тут найдётся 5 связных компонент. Теперь будем искать способ подсчёта связных компонент.

16.2. Ориентированный граф

Новый губернатор Банановых островов Гордон Крик удивился жалоб туземцев на транспортные проблемы. Накануне вступления в должность Гордон изучил карту архипелага, и точно знал, что после очередного тайфуна все острова, так или иначе, были связаны вновь построенными мостами (**рис. 14-1**).

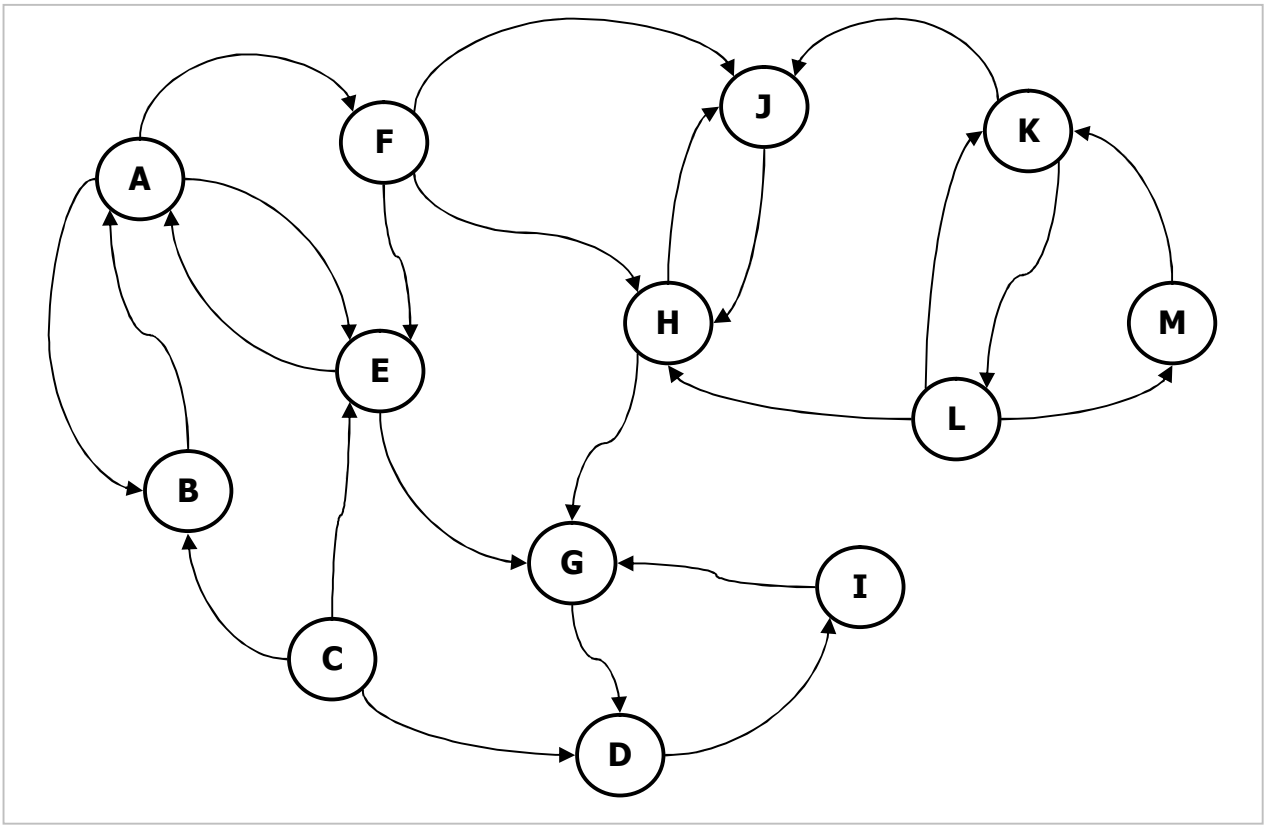


Рис. 16-2 — Мостовая сеть Банановых островов

Так оно и было, но губернатор упустил одну деталь: все мосты были узкими, и допускали лишь одностороннее движение. И оказалось, к примеру, что хотя житель острова *A* и мог доехать к своему дяде на остров *H* посуху, но возвращался морем на пароме. А к тёще, обитающей на острове *M*, он вообще не мог доехать по мостам (и она к нему тоже).

Сейчас вместе с группой «банановых» математиков займёмся решением мостовой проблемы. Наша задача: выделить группы островов (подмножества вершин) такие, что внутри каждой группы возможны взаимные поездки по мостам. Искомые подмножества вершин — это *сильные компоненты* графа. Получив все эти компоненты, мы сразу определим места для постройки недостающих мостов.

16.3. Связная компонента неориентированного графа

Возьмём произвольную вершину графа и составим множество всех *достижимых из неё* вершин, включая и саму эту вершину — мы назвали это множество *исходящей интегральной гаммой*. Понятно, что в неориентированном графе (где мосты двусторонние) это подмножество будет одной из связных

компонент. Алгоритм получения такой компоненты реализован в следующем методе.

Листинг 16-1 — Накопление исходящей интегральной гаммы

```
function TNode.GenGammaOut: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);       // начианем расширение гаммы с текущей вершины
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);           // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;    // начало перебора вершин
    while Assigned(Node) do begin
      Node.OutGammaAdd(Result);       // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;
```

Построение всех таких компонент будет рассмотрено далее.

16.4. Сильно связная компонента орграфа

Пусть мы определили в орграфе *исходящую* интегральную гамму, как это было показано выше. Достижима ли из всех вершин этой гаммы исходная вершина? Ясно, что в орграфе таких гарантий нет. Построим теперь для той же исходной вершины *входящую* интегральную гамму — множество вершин, из которых она достижима.

Листинг 16-2 — Накопление входящей интегральной гаммы

```
function TNode.GenGammaIn: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);       // начианем расширение гаммы с текущей вершины
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);           // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;    // начало перебора вершин
    while Assigned(Node) do begin
      Node.InGammaAdd(Result);        // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;
```

Теперь рассудим так: если какие-то вершины принадлежат одновременно обеим гаммам (а исходная вершина к ним точно принадлежит), то очевидно, что все такие вершины взаимно достижимы и образуют *сильную компоненту* графа. Вычисление такой компоненты реализовано в методе класса **TNode**:

Листинг 16-3 — Формирование сильной компоненты графа

```
function TNode.GenStrongArea: TSet;
var Temp: TSet;
begin
  Result:= GenGammaOut;                 // множество достижимых вершин
  if not mOwner.mDirect then Exit;      // выход, если это не орграф
  Temp:= GenGammaIn;                    // множество, из которых достижима исх.
  Result.Mul(Temp);                     // Результат = пересечение множеств
  Temp.Free;                            // освобождаем временное множество
end;
```

Стало быть, одна сильная компонента графа находится перемножением (пересечением) исходящей и входящей интегральной гамм любой из её вершин. Эта идея иллюстрируется следующими рисунками, где исходной выбрана вершина *A* . Тот же результат получится, если выбрать исходной вершину *B*, *E* или *F*.

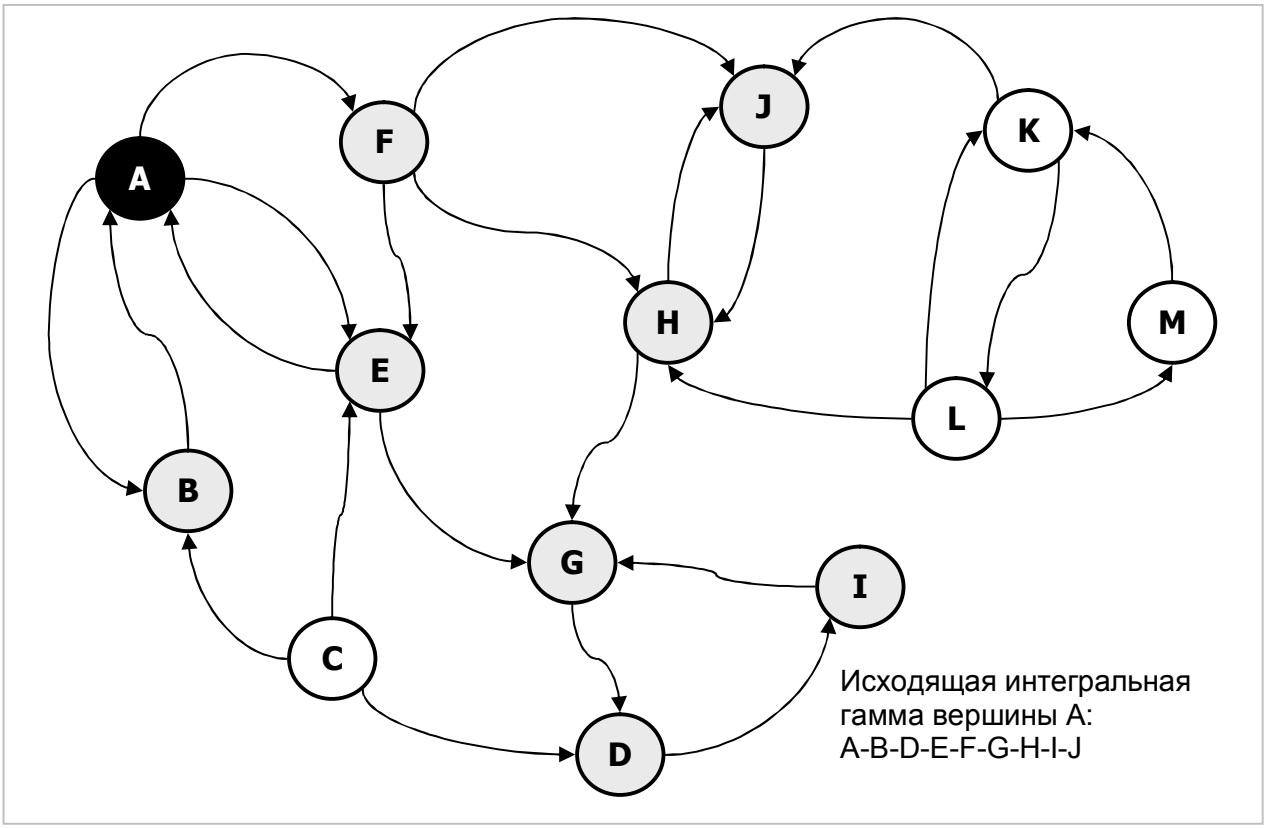


Рис. 16-3 — Вершины, достижимые из вершины A (исходящая гамма)

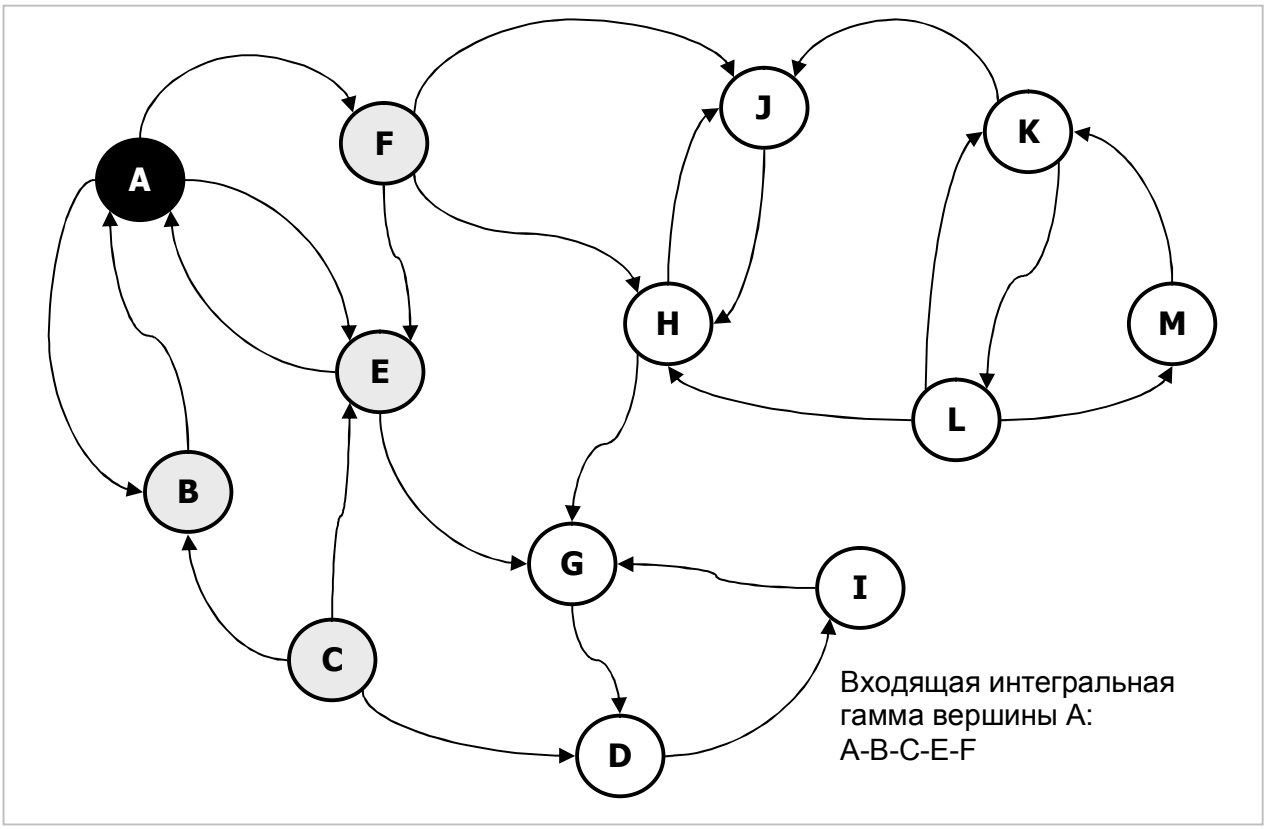


Рис. 16-4 — Вершины, из которых достижима вершина A (входящая гамма)

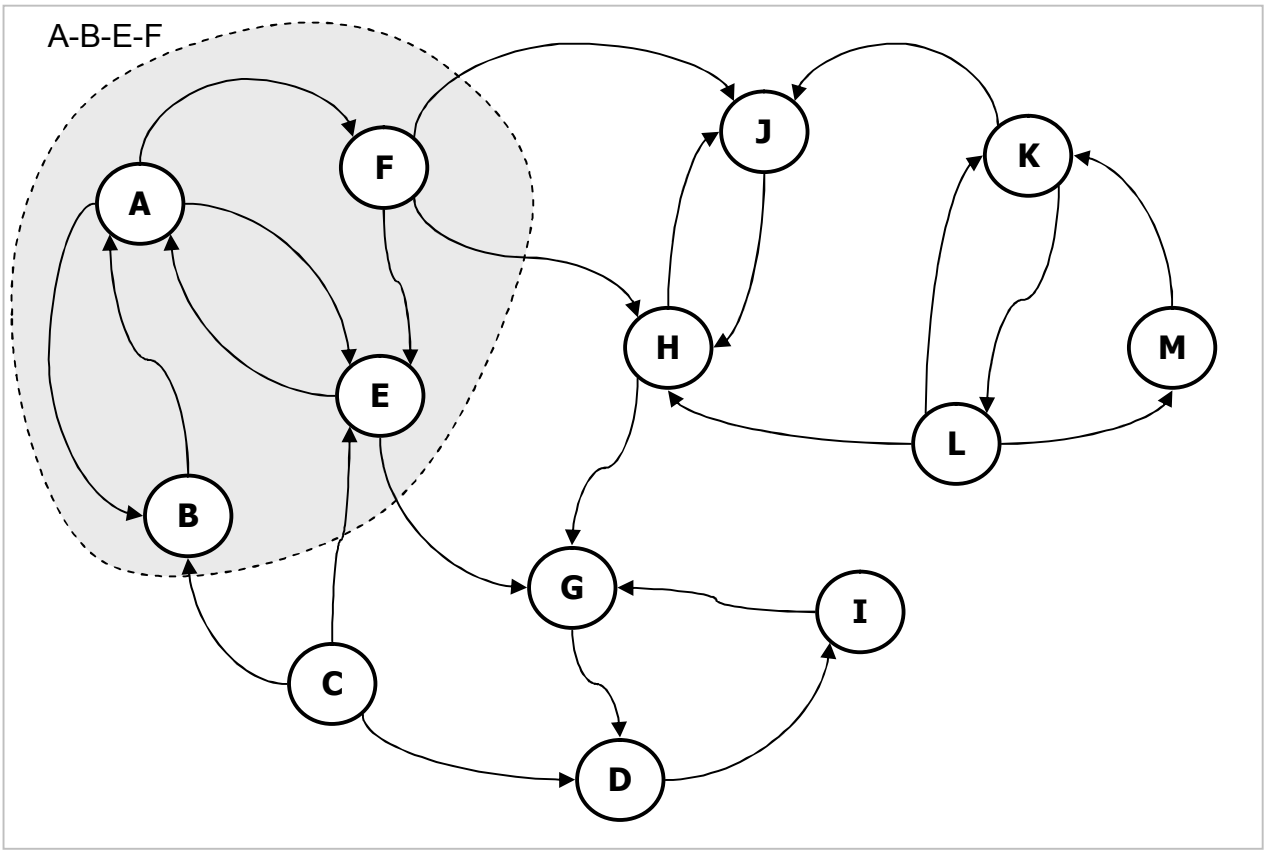


Рис. 16-5 — Сильная компонента A-B-E-F

16.5. Поиск всех сильных компонент

Итак, найдя *сильную компоненту связности* для некоторой вершины орграфа (или просто *компоненту связности*, если говорим о графе), мы можем подсчитать, и найти все такие компоненты. Идея проста. Создадим множество-копию всех вершин графа (копия будет постепенно очищаться). Затем из копии первую вершину и найдём соответствующую ей сильную компоненту. Удалив эту компоненту из копии, вновь возьмём из копии первую вершину и повторим те же действия, пока копия не очистится. Альтернативой копии может быть подходящая окраска вершин.

Листинг 16-4 — Подсчёт всех сильно связанных компонент

```
function TGraph.CalcAreas: integer;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
begin
  Result:= 0;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Inc(Result);                    // счётчик областей связности
    Node:= Copy.GetFirst as TNode;  // первая вершина из копии
    Strong:= Node.GenStrongArea;    // область её связности
    Copy.Sub(Strong);               // вычитаем из копии
    Strong.Free                     // освобождаем область
  until Copy.GetCount = 0;          // пока копия не пуста
  Copy.Free;
end;
```

Похожий метод возвращает множество всех связных областей в виде множества подграфов, порождённых на сильно связанных вершинах:

Листинг 16-5 — Порождение подграфов из сильно связанных компонент

```
function TGraph.GenAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
    Gr: TGraph;     // граф, порождённый на множестве вершин Strong
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Node:= Copy.GetFirst as TNode;  // первая вершина из копии
    Strong:= Node.GenStrongArea;    // область её связности
    Gr:= CopyByNodes(Strong);       // создание порождённого подграфа
    Result.Insert(Gr);              // вставка порождённого подграфа
    Copy.Sub(Strong);               // вычитаем область из копии
    Strong.Free                     // освобождаем область
  until Copy.GetCount = 0;          // пока копия не пуста
  Copy.Free;
end;
```

Здесь применён метод порождения подграфа из подмножества вершин:

Листинг 16-6 — Метод порождения подграфа из подмножества вершин

```
function TGraph.CopyByNodes(aNodes: TSet): TGraph;
var  N1, N2 : TNode;     // исходный узел и его копия
          ND : TNode;     // приёмник в копии
           L : TLink;
begin
  Result:= TGraph.Create('CopyByNodes', mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin   // если существует в множестве аргумента
      N2:= N1.Copy as TNode;         // то создаём копию
      Result.InsertNode(N2);         // и вставляем в граф
    end;
    N1:= NodeNext;
  end;
  // Копирование связей:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin   // если существует в множестве аргумента
      N2:= Result.GetNode(N1);       // узел копии
      L:= N1.OutLinkFirst;           // связь в исходном
      while Assigned(L) do begin     // перебор связей исходного
        ND:= Result.GetNode(L.mDest);      // приёмник в копии
        Result.SetLink(N2, ND, L.mValue); // связать узлы
        L:= N1.OutLinkNext;          // следующая связь
      end;
    end;
    N1:= NodeNext;    // следующий узел исходного
  end;
end;
```

Для проверки описанных выше методов предлагается следующая программа:

Листинг 16-7 — Подсчёт и формирование областей связности

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas';

var Gr : TGraphChars;
    N: integer;
    Areas: TSet;
    S: string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 1, 1, 20, 40);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - -');
    N:= Gr.CalcAreas;
    Writeln('Areas count= ', N);
    Writeln('- - - - - - - - - - - - - - - - - -');
```

```
    Areas:= Gr.GenAreas;
    if Assigned(Areas) then begin
      Areas.Expo;
      Areas.ClrAndDestroy;
      Areas.Free;
    end;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Gr.Free;
    Readln(S);
  until S<>'';
end.
```

16.6. Конденсация сильных компонент

Итак, с определением всех сильных компонент графа (рис. 16-6), то есть групп взаимно достижимых островов, нам открывается возможность найти такие острова, соединение которых мостами с односторонним движением полностью свяжет архипелаг. Разумеется, что из экономии число таких мостов должно быть минимально.

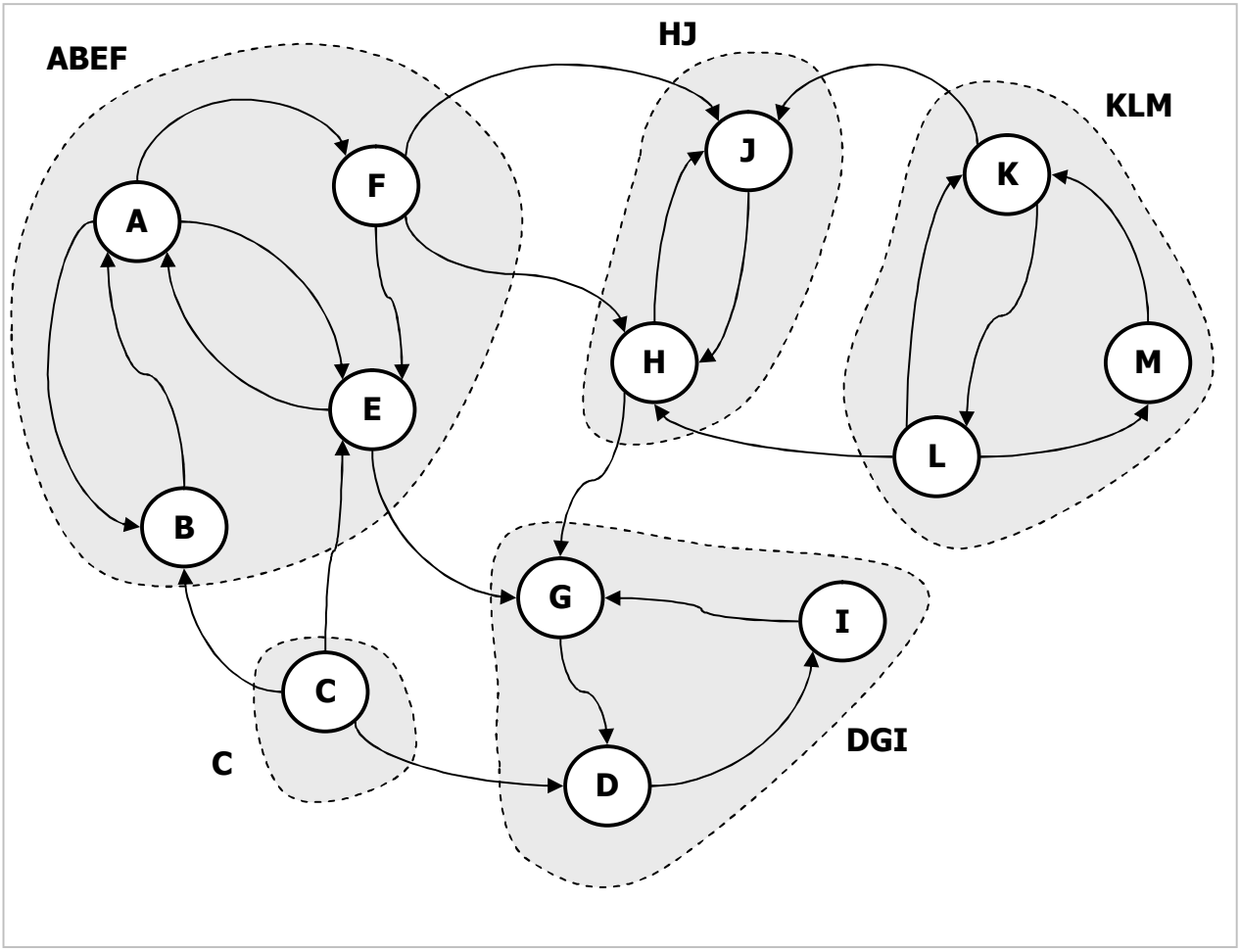


Рис. 16-6 — Сильные компоненты графа

В основе решения будет полученное нами множество сильных компонент, мы создадим из них новый граф. В этом графе сильные компоненты станут вершинами (не удивляйтесь, ведь и вершина, и множество — это наследники элемента **`TItem`**).

Этот новый тип вершины, содержащей сильную компоненту, описан через класс **TCondensNode**.

```
TCondensNode = class (TNode)
    public
      mName  : string;// имя = сумма входящих в компоненту имён вершин
      mNodes : TSet;  // собственные вершины
      mOut   : TSet;  // ссылки за пределы сильной компоненты (вершины)
      constructor Create(aNodes: TSet; aOwner: TGraph);
      destructor Destroy;  override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;
```

Связи между вершинами нового графа установим на основе связей исходного графа, но так, чтобы между двумя компонентами их было не более одной (то есть, без дублирующих связей). На рис. 16-7 они показаны пунктирными стрелками.

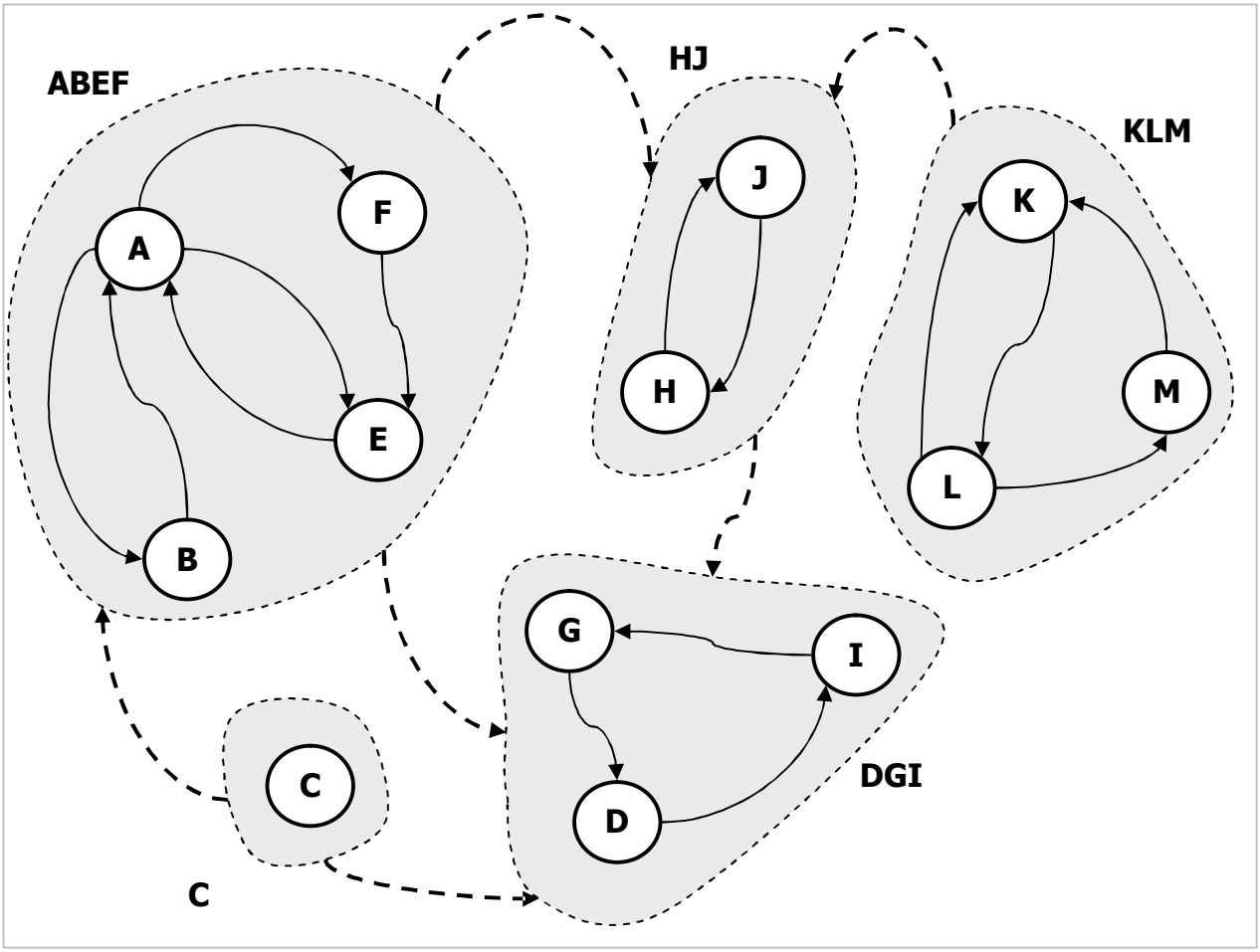


Рис. 16-7 — Подграфы, порождённые сильными компонентами

В результате построений у нас получился *конденсат* — граф, изображённый на рис. 16-8.

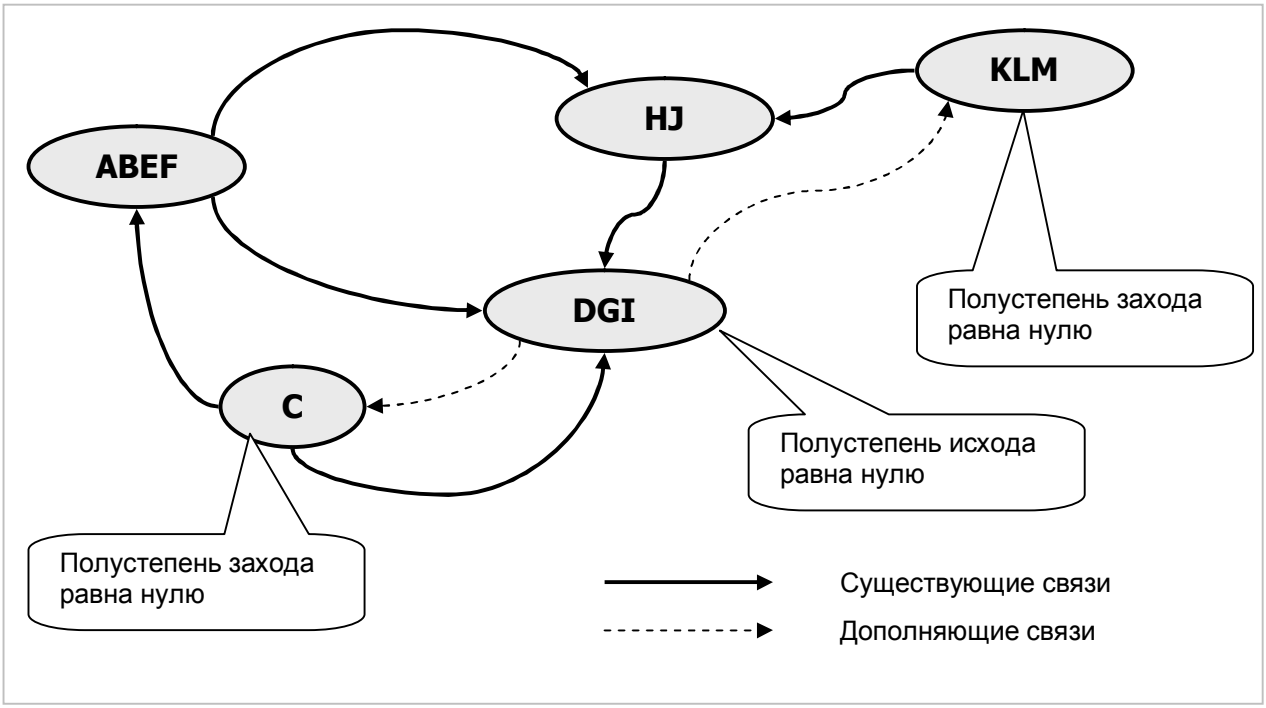


Рис. 16-8 — Конденсация сильных компонент

Здесь обратим внимание на полу-степени исхода и захода вершин. Вершины *C* и *KLM* имеют нулевые степени захода (нет входящих дуг), — значит попасть сюда из других вершин невозможно. Наоборот, вершина *DGI* имеет нулевую степень исхода (здесь нет входящих дуг) — это ловушка, из которой нет выхода. Как добиться полной связности архипелага минимальными средствами? Очевидно, надо построить исходящие связи (мосты), ведущие из ловушки *DGI* в недостижимые пока «медвежьи углы» *C* и *KLM*. Таким образом, достаточно построить не более двух мостов: между любыми островами из множества *D-G-I* к острову *C,* и между теми же островами и любыми из множества *K-L-M*. Эти дополняющие связи обозначены пунктиром. Вот методы, реализующие описанные выше алгоритмы.

Листинг 16-8 — Формирование множества-конденсата

```
function TGraph.GenStrongAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    N : TNode;
    Strong: TSet;  // сильная компонента
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Пока копия не пуста:
  while Copy.GetCount <> 0 do begin
    N:= Copy.GetFirst as TNode;       // первая вершина в копии
    Strong:= N.GenStrongArea;         // находим сильную компоненту
    Result.Insert(Strong);            // и вставляем в результат
    Copy.Sub(Strong);                 // удаляем её из копии
  end;
  Copy.Free;  // освобождаем копию
end;
```

Следующий метод пользуется предыдущим и строит новый граф — конденсат исходного графа.

Листинг 16-9 — Создание графа-конденсата

```
function TGraph.GenCondens: TGraph;
var Strongs : TSet;          // множество сильных компонент
          S : TSet;          // вспомогательное множество
         CN : TCondensNode; // текущий узел-конденсат
     Target : TCondensNode; // целевой узел-конденсат
begin
  Result:= TGraph.Create('Condens', true, false, false);  // пустой граф
  Strongs:= GenStrongAreas;                 // формируем множество-конденсат
  S:= Strongs.GetFirst as TSet;             // подмнож. вершин в узле конденс.
  while Assigned(S) do begin                // перебор множеств конденсата
    CN:= TCondensNode.Create(S, Result);  // создаём вершину конденсата
    Result.InsertNode(CN);                  // и вставляем в граф-конденсат
    S:= Strongs.GetNext as TSet;            // подмнож. вершин в узле конденс.
  end;
  // формирование связей между вершинами конденсата:
  S:= CreateSet;               // вспомогат. множество вершин исходного графа
  CN:= Result.NodeFirst as TCondensNode;        // вершина конденсата
  while Assigned(CN) do begin                   // перебор вершин конденсата
    if CN.mOut.GetCount <> 0 then begin         // если степень исхода <> 0
      Result.PosPush;                           // сохр. позицию перебора
      Target:= Result.NodeFirst as TCondensNode;// вершина конденсата
      while Assigned(Target) do begin           // внутр. цикл перебора
        if CN <> Target then begin              // если не текущая вершина
          S.CopyItems(CN.mOut);                 // исходящие связи текущей
          S.Mul(Target.mNodes);                 // пересекаем с целевыми
          if S.GetCount<>0                      // если связаны в исх. графе
            then Result.SetLink(CN, Target, 1); // то уст. связь в конденсате
        end;
        Target:= Result.NodeNext as TCondensNode; // след. узел конденсата
      end;
      Result.PosPop;                        // восст. позицию перебора
    end;
    CN:= Result.NodeNext as TCondensNode;  // след. узел конденсата
  end;
  S.Free;
  Strongs.ClrAndDestroy;  // очищаем множество-конденсат
  Strongs.Free;
end;
```

Для лучшего понимания представленных методов ознакомьтесь с устройством узла-конденсата **TCondensNode** по к листингу модуля **Graph**. Следующая программа служит для проверки метода постройки конденсата.

Листинг 16-10 — Тестирующая программа для проверки формирования конденсата

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas';
```

```
var Gr : TGraphChars;
    Condens: TGraph;  // граф-конденсат
    S: string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(true, 1, 1, 20, 40);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    Condens:= Gr.GenCondens;
    Condens.Expo;
    Condens.Free;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    Gr.Free;
    Readln(S);
  until S<>'';
end.
```

16.7. Анализ информационных потоков

В книге [7] дано интересное исследование информационных потоков посредство конденсата. Пусть карта изображает не мосты, а пути распространения информации между островами. В конденсированном виде этот граф представлен на рис. 16-9.

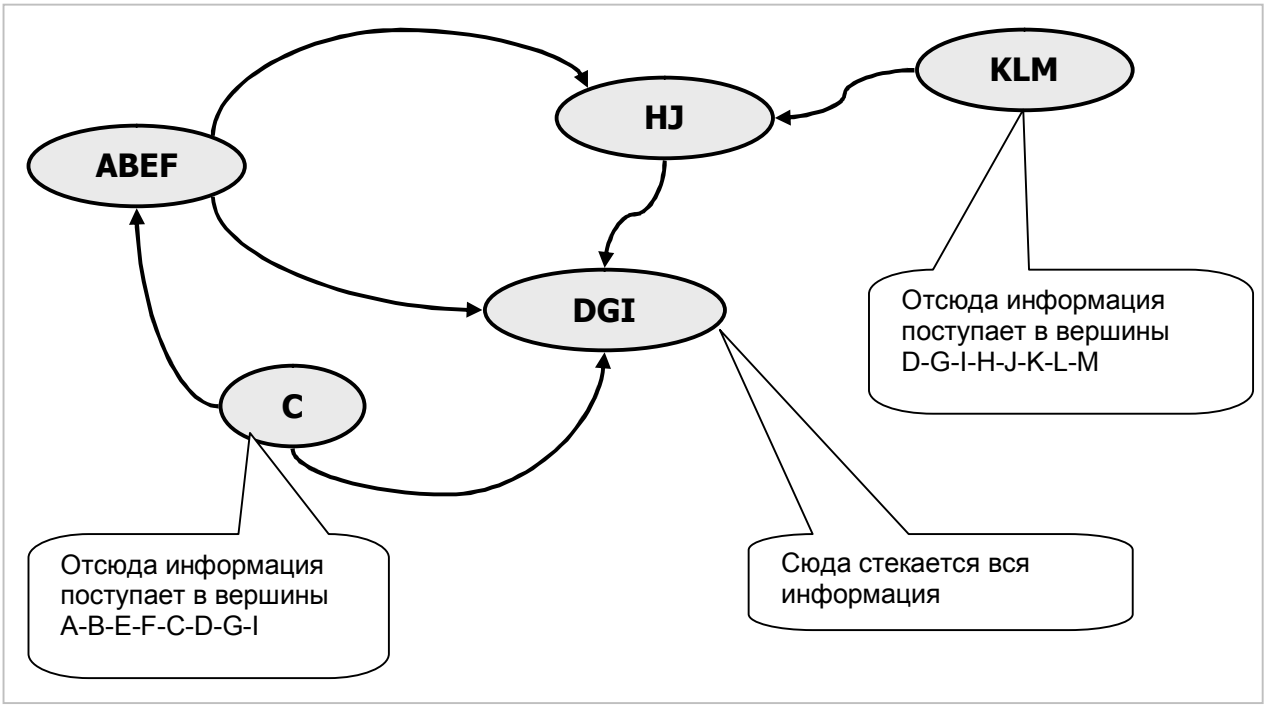


Рис. 16-9 — Конденсированный граф путей распространения информации

То есть, внутри конденсатов острова обмениваются информацией без ограничения, а за их пределами — по указанным на рисунке дугам. Пусть информацией будут некие управляющие команды, и надо выбрать минимальное подмножество островов, из которых команды могут попасть ко всем остальным. Выберем вершины конденсата с нулевыми степенями захода: это *C* и *KLM*. Из первой достижимы восемь вершин: **A-B-E-F-C-D-G-I**, из второй другие восемь: **D-G-I-H-J-K-L-M**. В итоге это даёт все вершины графа, стало быть, для

управления архипелагом (или рассылки спама) достаточно взять остров *C* и любой из островов *K*, *L* или *M*. Полученные подмножества называют *базами* графа, в данном графе имеется три базы: **C-K**, **C-L** и **C-M**.

Теперь представим, что ценные данные добываются в вершинах графа и распространяется теми же путями. Вершина *DGI* с нулевой степенью исхода отличается тем, что сюда стекается вся информация, вырабатываемая в вершинах графа. Таким образом, центр сбора и обработки информации надо разместить на одном из островов *D*, *G*, или *I*. Минимальные множества вершин, достижимых из любой вершины графа называют *антибазой* графа. Стало быть, в данном графе три антибазы, каждая из них состоит из одной вершины.

Пусть теперь граф изображает потоки информации и команд в некой организации, здесь вершины — это люди. Тогда любая из баз даст минимальный круг *могущественных* лиц, способных управлять всеми другими. Одной из антибаз станет минимальный круг вполне *осведомлённых* лиц. Пусть мы хотим составить управляющий *комитет* такой, чтобы в него входили в минимальном числе и *осведомлённые* и *могущественные* лица. Очевидно, что для этого надо объединить одну из баз и одну из антибаз. В данном случае мы сможем сформировать девять вариантов таких комитетов (3x3).

16.8. Итоги

16.8.1. Подмножество частично достижимых вершин орграфа называется *связной* компонентой.

16.8.2. Подмножество *взаимно* достижимых вершин называется *сильной* компонентой орграфа.

16.8.3. В неориентированном графе вершины одной связной компоненты взаимно достижимы, для таких графов *связная* компонента и *сильная* — одно и то же.

16.8.4. Сильная компонента орграфа находится пересечением подмножеств исходящей и входящей интегральных гамм некоторой вершины. Все такие компоненты находятся последовательным исключением из полного множества вершин найденных сильных компонент.

16.8.5. Граф, вершинами которого являются сильные компоненты, называется *конденсатом*. Конденсат — мощное средство исследования орграфов.

16.9. Задания

16.9.1. Достижение сильной связности графа

Создайте метод, возвращающий минимальный набор из пар вершин, которые необходимо связать между собой для обеспечения связности графа.

16.9.2. Поиск баз, антибаз и комитетов

Постройте методы для формирования баз, антибаз и комитетов орграфа.

16.10. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
√ 6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 17
Независимые вершины и клики

Здесь будут рассмотрены *независимые вершины* и *клики* — два взаимно противоположных и тесно связанных понятия, имеющих отношение к вершинам *неориентированного* графа.

17.1. Знакомство

Кликой (*Clique* — фр.) называют тесно сплочённую группу людей. Вот сцена в электричке: тут и там оживлённо беседуют небольшие группы людей. Заметно, что все они хорошо знакомы, — это *клики*. Другие пассажиры молча смотрят в окно или читают, не пытаясь общаться с незнакомыми соседями, — они подобны *независимым вершинам*, о которых тоже пойдёт речь.

17.1.1. О кликах

Изобразим схему взаимных знакомств людей неориентированным графом (рис. 17-1). Здесь два лица знакомы друг с другом, если вершины связаны ребром. Так, к примеру, у гражданина *A* нет знакомых, а гражданин *E* знаком с гражданами *C* и *D*, но не знаком с остальными.

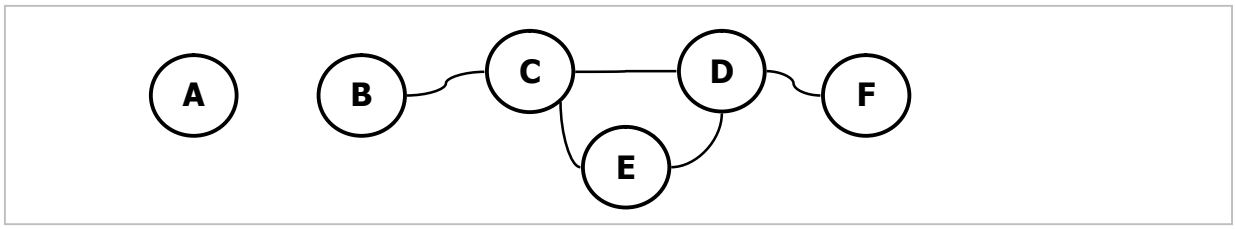


Рис. 17-1— Схема взаимных знакомств

Окажись в одном вагоне лица *C*, *D* и *E*, они образовали бы клику — подмножество вершин, в котором все вершины непосредственно связаны друг с другом (рис. 17-2).

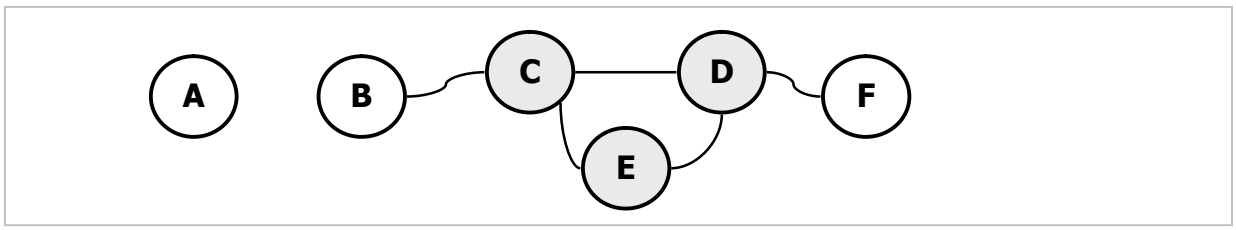


Рис. 17-2 — Клика C-D-E

В отсутствие лица *E*, могли бы образоваться менее мощные клики: *B-C*, *C-D* или *D-F*. Отметим, что тройки *B-C-D* или *C-D-F* не могут быть кликами, поскольку в первой не знакомы между собой *B* и *D*, а во второй — *C* и *F*.

Подобие клики — следующая в тыл противника группа фронтовых разведчиков. Будь вы командиром такой группы, то взяли бы к себе хорошо знакомого бойца. Третьего вы возьмёте при условии, что он знаком вам обоим, а четвёртого — при условии, что он знаком всем троим, и т.д.

Поищем все возможные клики графа, а точнее, все возможные *максимальные* клики. Максимальными они будут в том смысле, что такие клики уже нельзя расширить за счёт других (соседних) вершин. При постройке всех максимальных клик надо перепробовать в качестве начальных вершин («командиров») все вершины графа, сделаем это в алфавитном порядке.

Первую клику начнём строить с вершины *A*. Поскольку она изолирована, то останется единственной в этой клике. Вторую клику строим от вершины *B*, к ней можно присоединить *C*, и на этом расширение клики *B-C* закончится. Сборку третьей клики начнём с *C*, очевидно, что далее к ней присоединятся *D* и *E*, образуя клику *C-D-E*. Наконец, начав постройку с вершины *D*, получим клику *D-G*. В итоге получим четыре максимальных клики:

- *A*
- *B-C*
- *C-D-E*
- *D-F*

Все они максимальны в том смысле, что не могут быть расширены. В то же время, видно, что клика *C-D-E* является среди них максимальной по мощности. Отметим и то, что ни одно из этих множеств не является подмножеством другого (иначе оно не было бы максимальным).

17.1.2. О независимых вершинах

Подмножеству независимых вершин (*независимых множеств*, как их часто называют), так же, как и кликам, можно найти пример из разведки. Пусть вам поручили сплести глубоко законспирированную агентурную сеть. В целях безопасности вы вербуете людей так, чтобы никто из них не был знаком с другим участником сети: тогда, в случае провала, никто никого не выдаст. Проделаем эту работу с тем же множеством (рис. 17-1).

Для начала возьмём в нашу сеть вершину *A*. Поскольку она «не знакома» с другими вершинами, сеть можно расширить за счёт любой из оставшихся вершин, например, *B*. Теперь видно, что вершина *C* — сосед *B* — не годится для расширения сети, но можно «завербовать» *D*, *E* или *G*. Продолжим «вербовку» в алфавитном порядке, и присоединим *D*. С этого момента дальнейшее расширение сети стало невозможным, поскольку оставшиеся вершины (*C*, *E*, *F*) «знакомы» хотя бы с одним участником сети. Так мы построили одно из максимальных подмножеств независимых вершин: *A-B-D* (рис. 17-3).

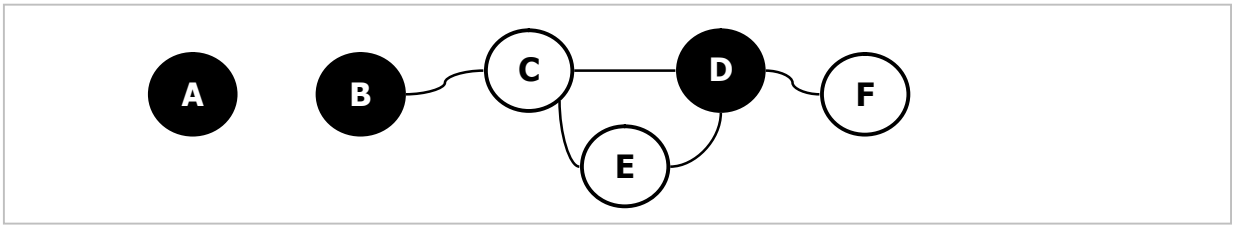


Рис. 17-3 — Независимые вершины A-B-D

Другие максимальные подмножества независимых вершин строятся аналогично. Например, после *A* можно присоединить не *B*, а *C* и далее *F*, в результате получится другое подмножество: *A-C-F*. На этом небольшом графе легко найти все такие подмножества:

- *A-B-D*
- *A-C-F*
- *A-B-E-F*

Поскольку вершина *A* изолирована, она входит в каждое из этих максимальных независимых подмножеств. Подобно максимальным кликам, каждое из этих подмножеств не является подмножеством другого.

В заключение знакомства рассмотрим ещё два примера приложений клик и независимых множеств. Пусть графы на рис. 17-4 изображают группу островов, соединённых мостами с двусторонним движением.

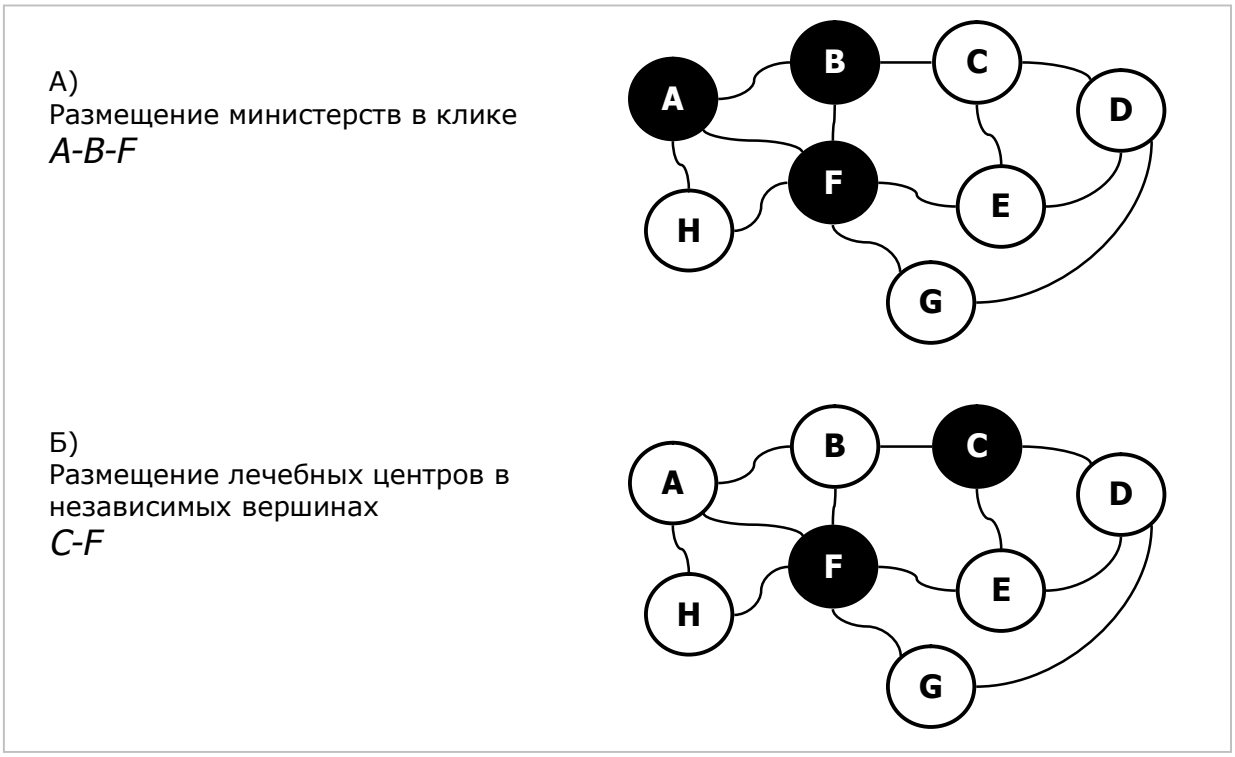


Рис. 17-4 — Размещение министерств (А) и лечебных центров (Б)

Испытывая недостаток места, правительство надумало разместить свои министерства на нескольких островах, но так, чтобы из любого из них можно было

добраться до любого другого, преодолев не более одного моста. Очевидно, что для этого подходит одна из двух максимальных клик: *A-B-F* или *C-D-E* (рис. 17-4 А).

Когда же правительство размещало лечебные центры, для удобства жителей стремились сделать так, чтобы пациенты преодолевали не более одного моста. Тут подходит одно из независимых множеств, имеющее наименьшую мощность, например *C-F*. Так обошлись минимальным количеством центров (рис. 17-4 Б).

17.1.3. Связь между кликами и независимыми вершинами

Говорят, что крайности сходятся. Это справедливо для независимых множеств и клик, которые связаны между собой через понятие *обратного графа*. Напомню, что граф, обратный исходному, строится на тех же вершинах путём добавления связей, которых нет в исходном графе, и удаления связей, имеющихся в исходном. Пример такого перестроения дан на рис. 17-5.

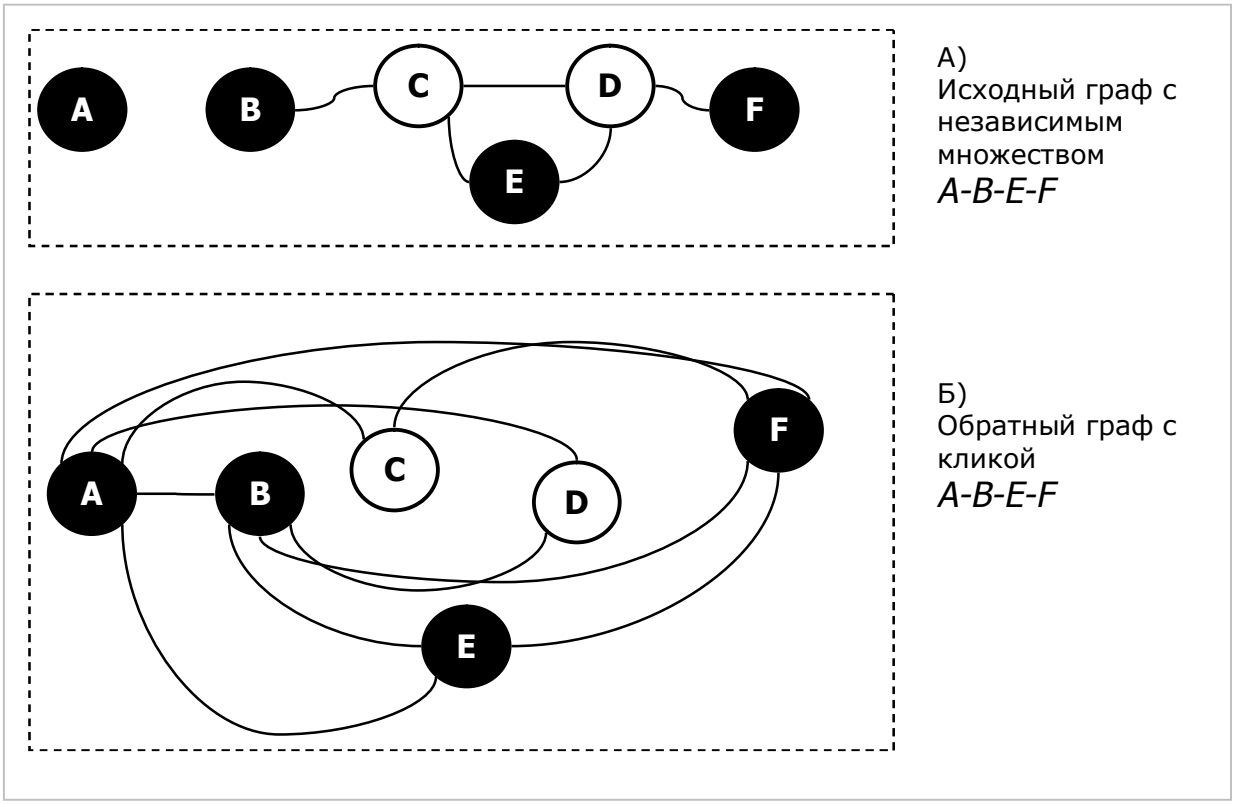


Рис. 17-5 — Исходный и обратный графы

Присмотревшись, можно заметить, что клики исходного графа становятся независимыми множествами вершин в обратном графе, и наоборот. Это сходство проявляется и в алгоритмах поиска всех максимальных *клик* и всех *независимых множеств*. По сути это две модификации одного алгоритма, придуманного голландцами Броном и Кербоши (*Bron* и *Kerbosh*).

17.2. Алгоритм Брона-Кербоша для поиска независимых вершин

Назначение алгоритма — сформировать множество максимальных подмножеств независимых вершин. Затем этот результат используется для выбора минимального или максимального по мощности подмножества.

17.2.1. Основная идея

Из предыдущих рассуждений видно, что все максимальные подмножества ищутся прямым перебором начальных вершин. Возьмём первую вершину, и вставим её в подмножество-результат, после чего удалим вершину и её непосредственных соседей из исходного множества. К оставшимся вершинам рекурсивно применим те же операции. Когда исходное множество опустеет, мы получим очередное максимальное подмножество независимых вершин. Выбрав на очередном уровне другую вершину-кандидат, мы двинемся по другой ветви перебора, и получим другое подмножество.

Можно догадаться, что для не сильно связанного графа количество ветвлений будет очень велико, и этот перебор будет чудовищно трудоёмок (NP-сложен).

17.2.2. Улучшение перебора

Брон и Кербош применили дополнительное условие, проверяемое при переходе к следующему уровню (метод ветвей и границ). Так они отсекли много тупиков, не ведущих к цели. В этом алгоритме на каждом уровне (шаге) используются три множества, которые мы обозначим так:

- **`Cand`** — вершины-кандидаты на включение в результат;
- **`Res`** — текущий результат;
- **`Tested`** — уже обработанные вершины.

Сначала в множество **`Cand`** помещаем все вершины графа, а прочие множества будут пусты. Затем из кандидатов выбираем первую вершину, включаем её в результат **`Res`**, и вместе с соседями вычёркиваем из кандидатов. Вдобавок её соседей вычёркиваем из обработанных вершин **`Tested`** — смысл этого вычёркивания прояснится позже. Перейдя к следующему уровню (при соблюдении двух условий, обсуждаемых ниже), повторим те же действия, и т.д., пока списки **`Cand`** и **`Tested`** не опустеют. Тогда в **`Res`** мы получим очередное независимое подмножество. Если же кандидаты будут исчерпаны, а в **`Tested`** остались не вычеркнутые вершины, это значит, что подмножество **`Res`** не максимально (в нём должно быть что-то из **`Tested`**, но не попало) — это тупик. Тогда (и в успешном, и в тупиковом случаях) выходим на предыдущий уровень, удаляя обработанную вершину из **`Res`** и **`Cand`**, и вставляя её в **`Tested`**.

Для отсечения тупиковых ветвей при каждом рекурсивном входе проверяем два условия: 1) исчерпание всех кандидатов и 2) граничат ли все обработанные

вершины в **Tested** с вершинами в **Cand**. Если хотя бы одна вершина не граничит, значит, **Tested** никогда не будет очищен, и тупик неизбежен. Обнаружив эту ситуацию, отбросим данную ветвь и вернёмся на уровень выше. Вот для чего соседи очередной вершины вычёркивались из **Tested**.

17.2.3. Пример

В табл. 17-1 дана частичная последовательность формирования независимых подмножеств для графа на рис. 17-1. После обработки вершины *A* и попадания её в множество **Tested** (шаг 2), выбор любой из оставшихся пяти вершин ведёт в конечном счёте в тупик. Так, на шаге 2-1-1 множество **Res=** *B-D* оказалось подмножеством *A-B-D*, поэтому шаги, отмеченные серым цветом, в действительности выполняться не будут.

Табл. 17-1 — Пример формирования независимых подмножеств

Шаг	Cand	Выбрано	Tested	Res	Примечание
1	A B C D E F	A	-	-	
1-1	B C D E F	B	-	A	
1-1-1	D E F	D	-	A B	
1-1-1-1	-	-	-	A B D	Готово
1-1-2	D E F	E	-	A B	
1-1-2-1	F	F	-	A B E	
1-1-2-1-1	-	-	-	A B E F	Готово
1-2	B C D E F	C	-	A	
1-2-1	F	F	-	A C	
1-2-1-1	-	-	-	A C F	Готово
2	B C D E F	B	A	-	
2-1	D E F	D	A	B	
2-1-1	-	-	A	B D	Тупик

17.2.4. Листинг

Ниже представлен листинг функции **TGraph.GenUndepend**, создающей все максимальные подмножества независимых вершин. Основу метода составляют рекурсивная локальная процедура **Extend** и локальная функция **Test**, проверяющая наличие независимых вершин среди уже проверенных.

Листинг 17-1 — Формирование максимальных подмножеств независимых вершин

```
function TGraph.GenUndepend: TSet;
var
  Res: TSet;     // очередное максимальное независимое множество
  Gamma: TSet;  // для доступа к соседним вершинам
  //-------------------------------------------------------------
  // Возвращает TRUE, если среди обработанных вершин (aTested)
  // есть хоть одна, не соседствующая ни с одним кандидатом (aCand)

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);   // первый из обработанных
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи обработанного узла
      Gamma.Mul(aCand);        // Gamma = пересечение соседей с кандидатами
      if Gamma.GetCount=0 then begin // если пересечение пусто,
        Result:= true;  // значит этот обработанный узел
        Break;          // не соседствует ни с одним кандидатом
      end;
      Node:= TNode(aTested.GetNext);  // следующий из обработанных
    end;
  end;
  //-------------------------------------------------------------
  // Рекурсивная процедура расширения независимого множества
  // aCand, aTested - множества с предыдущего уровня

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // новые вершины-кандидаты
    newTested: TSet; // новые обработанные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и
    // нет независимых от них обработанных вершин
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst);     // взять первого кандидата
      Res.Insert(Node);                 // вставить в результат
      Node.OutGammaGet(Gamma);          // Gamma= его соседи
      // Создать копии множеств кандидатов и проверенных
      newCand:= TSet(aCand.Copy);
      newTested:= TSet(aTested.Copy);

      newCand.Delete(Node);   // удалить текущий узел из кандидатов
      newCand.Sub(Gamma);     // и удалить его соседей
      newTested.Sub(Gamma);   // удалить соседей из копии проверенных

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдено очередное макс. независимое множество
             Result.Insert(Res.Copy)
        else if newCand.GetCount<>0 then
               // если остались кандидаты, то рекурсивный вызов
               // с новыми наборами кандидатов и проверенных
               Extend(newCand, newTested);
      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
```

```
      Res.Delete(Node);      // удалить текущий из результата
      aCand.Delete(Node);    // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end; // while
  end;
  //-------------------------------------------------------------
var
  Cand : TSet;    // вершины-кандидаты
  Tested: TSet;  // проверенные вершины

begin
  // Инициализация
  Result:= CreateSet;       // пустое множество-результат
  Res:= CreateSet;          // очередное макс. независимое множество
  Cand:= CreateSet;         // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested); // рекурс. процедура построения независимых множеств
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;
end;
```

17.2.5. Тестовый пример

Метод **GenUndepend** проверен на графе, входной файл которого показан ниже,

```
KRISTOFIDES Page 44
0 - граф(0), орграф(1)
0 - нагруженность вершин
0 - нагруженность рёбер (дуг)
8 - количество вершин
A B C D E F G H
A -> B F H
B -> A C F
C -> B D E
D -> C E G
E -> C D F
F -> A B E G H
G -> D F
H -> A F
```

Испытания дали следующие подмножества независимых вершин рис. 17-6):

```
A-C-G,  A-D,  A-E-G,  B-D-H,  B-E-G-H,  C-F,  C-G-H,  D-F
```

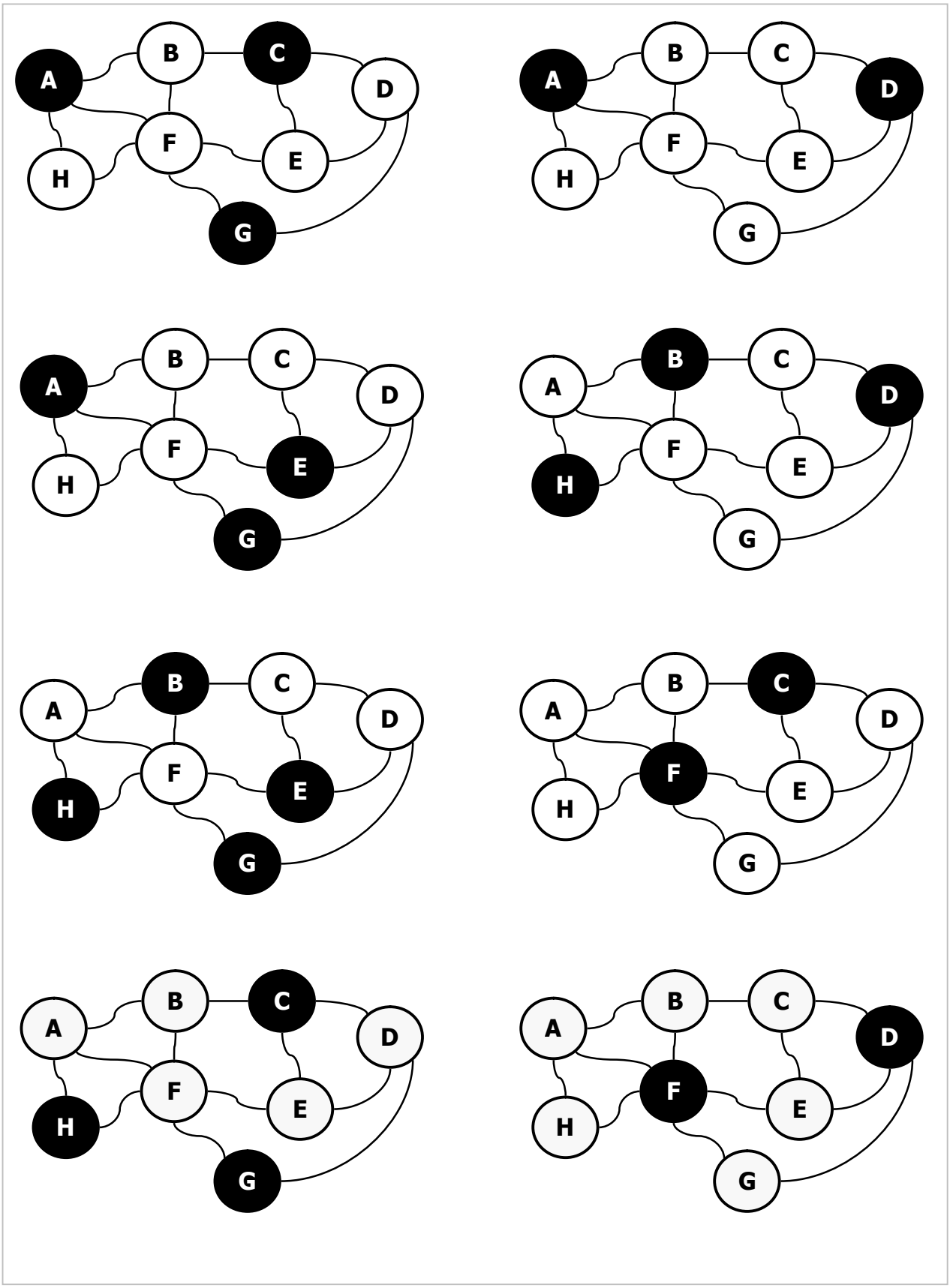


Рис. 17-6 — Независимые вершины графа

17.3. Алгоритм Брона-Кербоша для поиска максимальных клик

Назначение алгоритма — сформировать множество максимальных клик.

17.3.1. Основная идея

Взяв алгоритм поиска независимых множеств, и заменив в нём некоторые операции зеркально противоположными, мы получим то, что нам нужно. В этом следствие взаимной противоположности клик и независимых вершин. В табл. 17-2 показаны изменения, которые преобразуют один алгоритм в другой.

Табл. 17-2 — Отличия алгоритмов Брона-Кербоша

Процедура	Действия	
	Для независимых вершин	Для клик
Формирование новых кандидатов newCand	**Из кандидатов удаляются соседние вершины**	**В кандидатах остаются только соседние вершины**
Формирование новых проверенных newTested	**Удаляются соседние вершины**	**Остаются только соседние вершины**
Тестирование обработанных вершин на входе в процедуру расширения (функция Test)	**Функция Test проверяет наличие хотя бы одной вершины, не соседствующей ни с одним кандидатом**	**Функция Test проверяет наличие хотя бы одной вершины, соседствующей со всеми кандидатами**

17.3.2. Листинг

```
//      Формирование множества максимальных  клик
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Неориентированый граф

function TGraph.GenClique: TSet;
var
  Res: TSet;      // очередная клика
  Gamma : TSet;  // соседние вершины
  //-------------------------------------------------------------
  // Возвращает TRUE, если среди проверенных вершин в aTested
  // есть хоть одна, примыкающая ко всем кандидатами в aCand

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи Node
      Gamma.Mul(aCand);        // Gamma = соседи * кандидаты
      if Gamma.GetCount = aCand.GetCount then begin
        // если примыкает ко всем кандидатам
        Result:= true;
        Break;
      end;
      Node:= TNode(aTested.GetNext);
    end;
  end;
  //-------------------------------------------------------------
```

```
  // Рекурсивная процедура расширения клики

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // вершины-кандидаты
    newTested: TSet; // проверенные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и в проверенных aTested нет таких,
    // что примыкают ко всем кандидатами в aCand
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst); // очередной кандидат
      Res.Insert(Node);             // вставляем в клику
      Node.OutGammaGet(Gamma);      // Gamma:= соседи Node

      newCand:= TSet(aCand.Copy);   // копия кандидатов
      newCand.Delete(Node);         // удаляем текущую вершину
      newCand.Mul(Gamma);           // и оставляем только соседние

      newTested:= TSet(aTested.Copy); // копия проверенных
      newTested.Mul(Gamma);           // оставляем только соседние

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдена очередная клика
             Result.Insert(Res.Copy)

        else if newCand.GetCount<>0 then
             // если остались кандидаты, то рекурсивный вызов
             // с новыми наборами кандидатов и проверенных
             Extend(newCand, newTested);
      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
      Res.Delete(Node);     // удалить текущий из результата
      aCand.Delete(Node);   // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end
  end;
  //-------------------------------------------------------------
var
  Cand : TSet;   // вершины-кандидаты
  Tested: TSet;  // проверенные вершины
begin
  // Инициализация
  Result:= CreateSet;      // создать пустое множество-результат
  Res:= CreateSet;         // очередная клика
  Cand := CreateSet;       // кандидаты
  Cand.CopyItems(mNodes);  // Cand = все вершины графа
  Tested:= CreateSet;      // проверенные вершины
  Gamma:= CreateSet;       // соседние вершины

  Extend(Cand, Tested);  // рекурсивная процедура
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;
end;
```

17.3.3. Контрольный пример

Метод **GenClique** проверен на графе, входной файл которого показан ниже.

```
KRISTOFIDES Page 44
0 - граф(0), орграф(1)
0 - нагруженность вершин
0 - нагруженность рёбер (дуг)
8 - количество вершин
A B C D E F G H
A -> B F H
B -> A C F
C -> B D E
D -> C E G
E -> C D F
F -> A B E G H
G -> D F
H -> A F
```

Получены следующие подмножества максимальных клик (см. рис. 17-7):

```
A-B-F,  A-F-H,  B-C,  C-D-E,  D-G,  E-F,  F-G
```

17.4. О трудоёмкости

Здравый смысл подсказывает (а опыт подтверждает), что в разреженном графе клики должны формироваться быстро, поскольку у вершин соседей немного, и круг кандидатов в клику быстро редеет. Наоборот, поиск независимых множеств в таком графе будет затруднён, и тут сильно выручает отсечение тупиковых ветвей функцией **Test**, что убыстряет перебор на много порядков. В плотном графе всё наоборот (этого следовало ожидать): клики ищутся долго, а независимые вершины быстро.

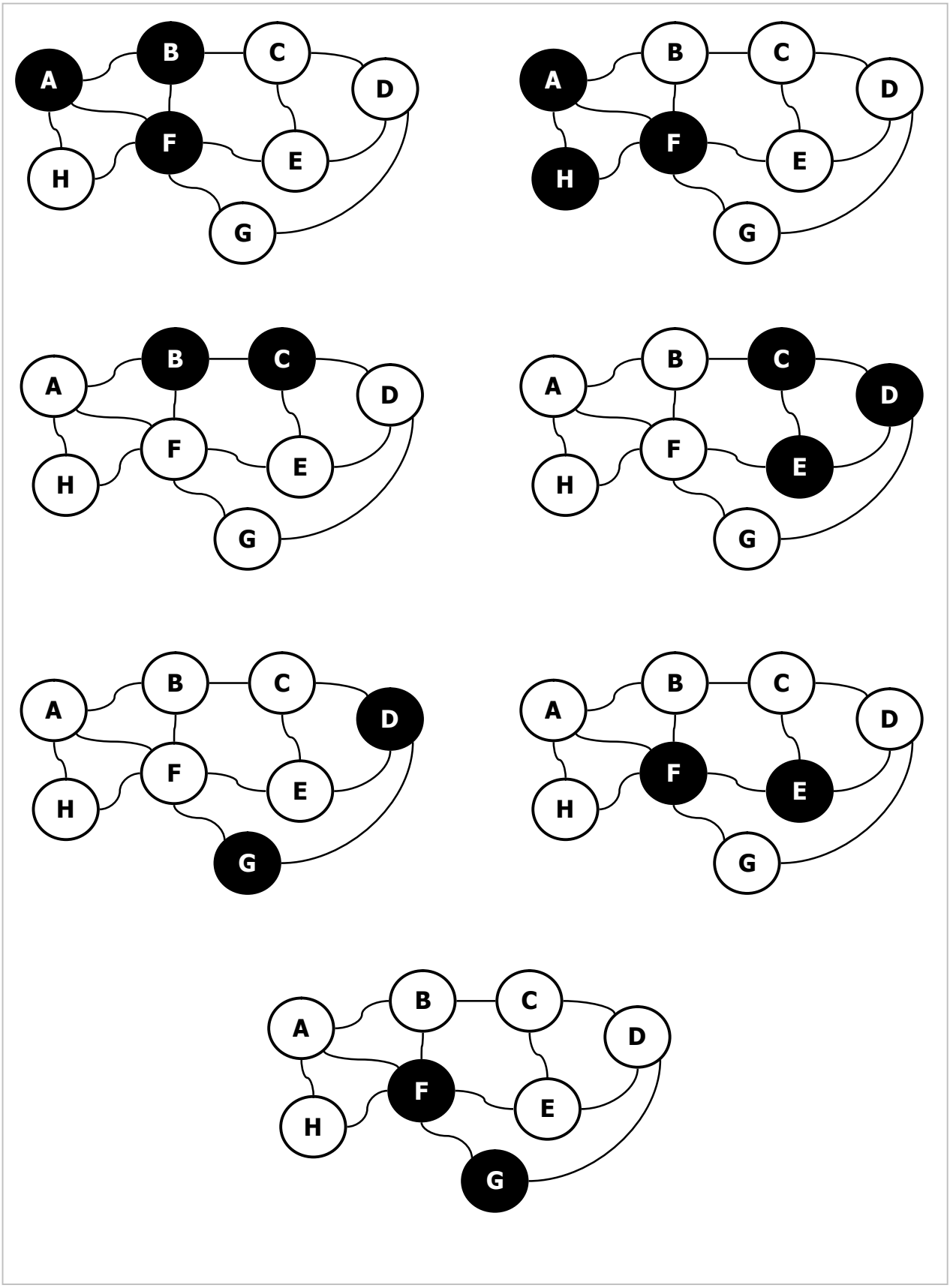


Рис. 17-7 — Клики графа

17.5. Итоги

17.5.1. Клики и независимые вершины — это два взаимно противоположных и тесно связанных понятия.

17.5.2. Клики — это подмножества вершин, где каждая вершина связана с каждой в этом же подмножестве.

17.5.3. Независимые множества — это такие подмножества вершин, где никакая вершина не связана с другой из этого подмножества.

17.5.4. Клики обратного графа совпадают с независимыми множествами исходного графа, и наоборот.

17.5.5. Для поиска клик и независимых вершин используют две модификации алгоритма Брона-Кербоша.

17.6. Задания

17.6.1. Межплеменные трения

Когда острова Бананового архипелага соединились мостами, выявились трения между племенами некоторых островов. Предложите способ расселения туземцев по островам так, чтобы ни одна пара взаимно конфликтующих племён не проживала на островах, непосредственно соединённых мостами.

17.6.2. Миротворческие силы

Пусть конфликтующие племена расселены в соответствии с решением задачи 17.6.1. Предложите подмножество островов, где желательно разместить миротворческие силы, препятствующие взаимным контактам враждующих племён через третьи острова.

17.7. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 43
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 18
Минимальные доминирующие множества

Найдя в предыдущей главе *множества независимых вершин* графа, мимоходом мы решили и задачу оптимального размещения лечебных центров: для этого годится одно из независимых множеств минимальной мощности. Рассмотрим более общую задачу этого рода: поиск оптимального размещения экстренных служб на любых графах, включая графы с взвешенными вершинами и дугами.

18.1. Постановка задачи

Рассмотрим острова, соединенные мостами с односторонним движением так, как показано на рис. 18-1.

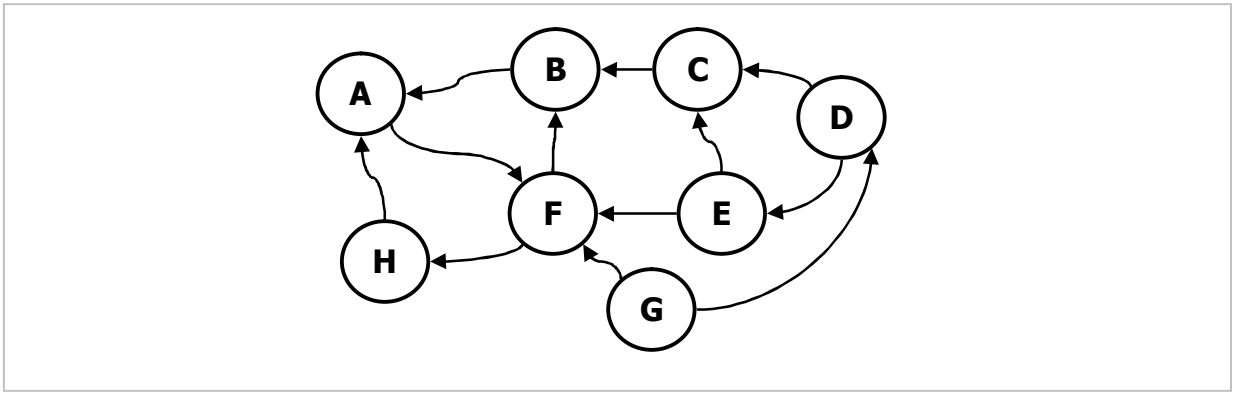


Рис. 18-1 — Островной архипелаг с односторонними мостами

Будем искать острова для размещения пожарных команд с тем, чтобы от места базирования до места пожара их автомобили преодолевали не более одного моста (они должны тушить свой остров и ближайших соседей). Обратите внимание, что возвращаться пожарные могут либо окружными путями, либо морем — обратный маршрут нас не интересует.

У задачи есть вариации. Например, стоимость постройки пожарной части и её содержание на разных островах может отличаться, и тогда ищут вариант минимальной стоимости проекта (на графе с взвешенными вершинами). В другом случае может быть важной длина мостов (веса дуг), и тогда минимизируют время их преодоления. Возможна и комбинация этих вариантов. Подмножество находимых здесь вершин называют *минимальным доминирующим множеством*.

18.2. Доминирующие множества

В графе может быть несколько доминирующих множеств, все они характерны тем, что прочие вершины достигается из этого множества не более чем за шаг. Но нас будет интересовать только одно из этих доминирующих множеств, имеющее либо минимальную мощность (содержащее наименьшее количество вершин), либо минимальную стоимость (`Cost`) с учётом весов вершин и/или дуг.

Результат нашего поиска будет зависеть от того, ориентирован ли граф, взвешены ли его вершины и дуги, но все модификации задачи решаются одной общей и несложной процедурой.

18.3. Принцип решения

Рассмотрим следующую таблицу.

Табл. 18-1 — Подмножества покрывающих граф вершин

Корневая вершина (откуда)	Подмножества достигаемых вершин (куда)							
A	A	-	-	-	-	F	-	-
B	A	B	-	-	-	-	-	-
C	-	B	C	-	-	-	-	-
D	-	-	C	D	E	-	-	-
E	-	-	-	-	E	F	-	-
F	-	-	-	-	-	F	-	H
G	-	-	-	D	-	F	G	-
H	A	-	-	-	-	-	-	H
Сумма подмножеств	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Табл. 18-1 построена путём перебора вершин графа следующим образом. Взяв очередную вершину (назовём её корневой), поместим её в левую колонку, а в правых столбцах отметим вершины, доступные из корневой не более чем за шаг (сюда попадает и корневая). Так справа оказались подмножества достигаемых вершин, в сумме покрывающие все вершины графа. Остался сущий пустяк: собрать из подмножеств достигаемых вершин минимальное покрытие, — эта стандартная задача на множествах (ЗНП) решена нами в главе 11.

18.4. Листинг

Задача поиска одного из минимальных доминирующих множеств решается методом **`GenDominating`**, он представлен ниже. Своей простотой он обязан функции **`CollectMinCover`** из модуля **`Assembly`**. Немало потрудившись при создании этой изощрённой функции (глава 11), мы извлекаем теперь законную прибыль. Здесь мы заняты главным образом формированием промежуточного буфера, в который заносим оценённые подмножества достигаемых вершин (табл. 18-1).

Листинг 18-1 — Формирование доминирующего множества (Граф + орграф)

```
function TGraph.GenDominating: TCostSet;
var  Buf: TBuffer;    // промежуточный буфер
     Node: TNode;       // Вершина графа
     Gamma: TSet;       // для множества соседей
     CS: TCostNodes;    // оценённое подмножество
begin
  Buf:= TBuffer.Create;
  // В промежуточный буфер заносим оценённые подмножества,
  // каждое из которых соответствует одной вершине
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Gamma:= CreateSet;         // создаём пустое множество
    Node.OutGammaGet(Gamma);   // Gamma= соседние вершины
    Gamma.Insert(Node);        // + текущая
    // Node.CalcCost вычисляет цену узла как сумму веса узла и веса дуг
    CS:= TCostNodes.Create(Node.CalcCost, Gamma, Node);
    Buf.Push(CS);          // заносим в буфер
    Node:= NodeNext;
  end;
  // Поиск минимального покрытия
  Result:= Assembly.CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  SetUtils.RemoveItemsFromBuf(Buf, Result.mSet);
  // а буфер очищаем и уничтожаем
  Buf.ClrAndDestroy;
  Buf.Free;
end;
```

В промежуточный буфер заносятся элементы класса **TCostNodes** — это потомок класса **TCostSet** с добавленным полем — корневой вершиной, здесь также переопределены конструктор и метод вывода в текстовый файл.

```
// Элемент для представления доминирующих множеств
type
  TCostNodes = class (TCostSet)
    mNode: TNode;  // вершина-элемент доминирующего множества
    constructor Create(aCost: integer; aSet: TSet; aNode: TNode);
    procedure   Print(var aFile: TextFile); override;
  end;

constructor TCostNodes.Create(aCost: integer; aSet: TSet; aNode: TNode);
begin
  inherited Create(aCost, aSet, true);
  mNode:= aNode;
end;

procedure TCostNodes.Print(var aFile: TextFile);
var Node: TNode;
begin
  Write(aFile, mNode.GetName, ' = ', mCost, ' -> { ');
  Node:= TNode(mSet.GetFirst);
  while Assigned(Node) do begin
    Write(aFile, Node.GetName + ' ');
    Node:= TNode(mSet.GetNext);
  end;
  Writeln(aFile, '}');
end;
```

«Ценовую политику» при создании подмножеств определяет метод **Node.CalcCost**, вычисляющий цену соответствующего подмножества на основе полей **mValue** в объектах-вершинах и объектах-связях. Если же вершины и дуги графа не взвешены, то каждому подмножеству назначается единичная цена, и тогда алгоритм ищет подмножество вершин минимальной мощности.

```
// Подсчёт стоимости узла для поиска минимальных покрытий

function TNode.CalcCost: integer;
var L: TLink;
begin
  Result:=1;
  if mLoadNodes then Result:= mValue;
  if not mLoadLinks then Exit;
  // добавляем суммарную стоимость исходящих дуг (рёбер)
  L:= OutLinkFirst;
  while Assigned(L) do begin
    Inc(Result, L.mValue);
    L:= OutLinkNext;
  end;
end;
```

Наконец, перед выходом из метода **GenDominating** очищается и удаляется промежуточный буфер, но так, чтобы не потерять подмножества, попавшие в результат. Для этого вызывается процедура **RemoveItemsFromBuf** из модуля **SetUtils**.

18.5. Контрольные примеры

Ниже представлены результаты испытаний метода на ориентированном графе (рис. 17-1) и на том же графе с взвешенными вершинами (рис. 18-3). Во втором примере вес вершины *F* задан равным 9, в результате чего она потеряла статус доминирующей, передав «полномочия» двум соседним.

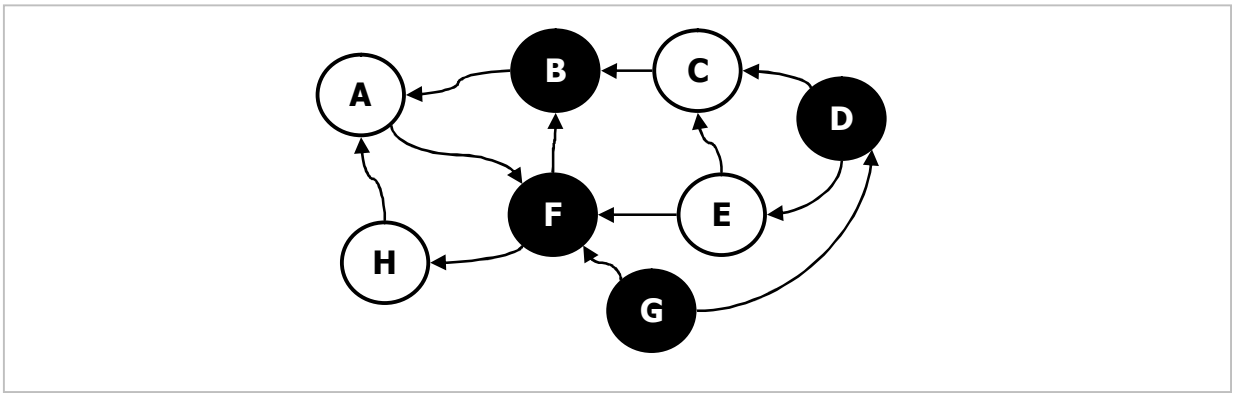


Рис. 18-2— Доминирующее множество не взвешенного графа

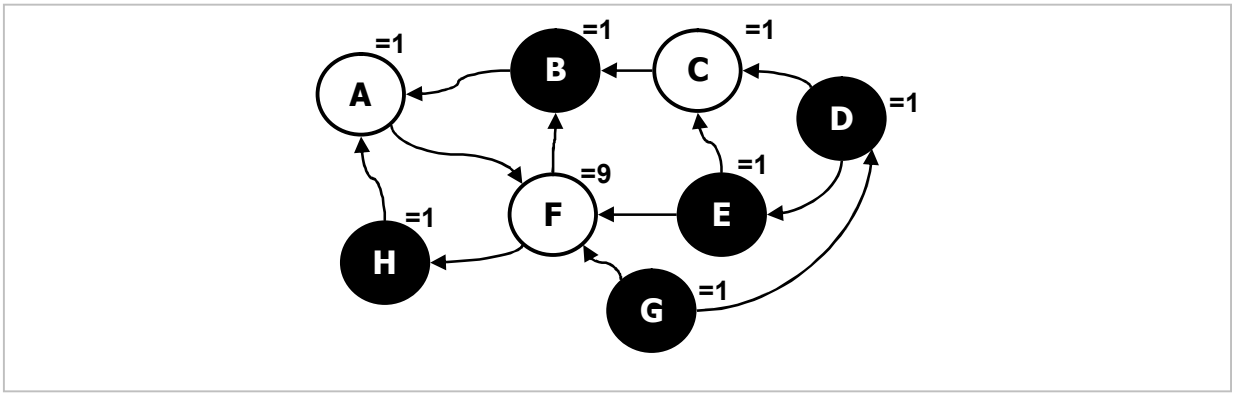


Рис. 18-3 — Доминирующее множество графа с взвешенными вершинами

18.6. Итоги

18.6.1. Путь из доминирующего множества вершин в любую другую пролегает не более чем через одну дугу (ребро).

18.6.2. На практике интерес представляют минимальные доминирующие множества, они отыскиваются через решение задачи о минимальном покрытии (ЗНП).

18.6.3. Минимальные доминирующие множества могут разыскиваться либо с учётом, либо без учёта весов вершин и дуг.

18.7. Задания

18.7.1. Проверка на разных графах

Проверьте работу алгоритма:

- на графе с взвешенными дугами;
- на не ориентированном графе.

Совпадут ли во втором случае минимальное доминирующее множество с одним из независимых множеств?

18.7.2. Доминирующие множества «второй степени»

Разработайте метод нахождения доминирующих множеств «второй степени», то есть таких вершин, из которых любая вершина графа достигается не более чем через две дуги.

18.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 43
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 19
Раскраски

19.1. Постановка задачи

Обратимся к «детской забаве»: раскрасим вершины неориентированного графа минимальным набором красок так, чтобы ни у одной вершины не было соседки того же цвета (чтобы в любой паре инцидентных вершин цвета их отличались). Пример такой раскраски в три цвета показан на рис. 19-1.

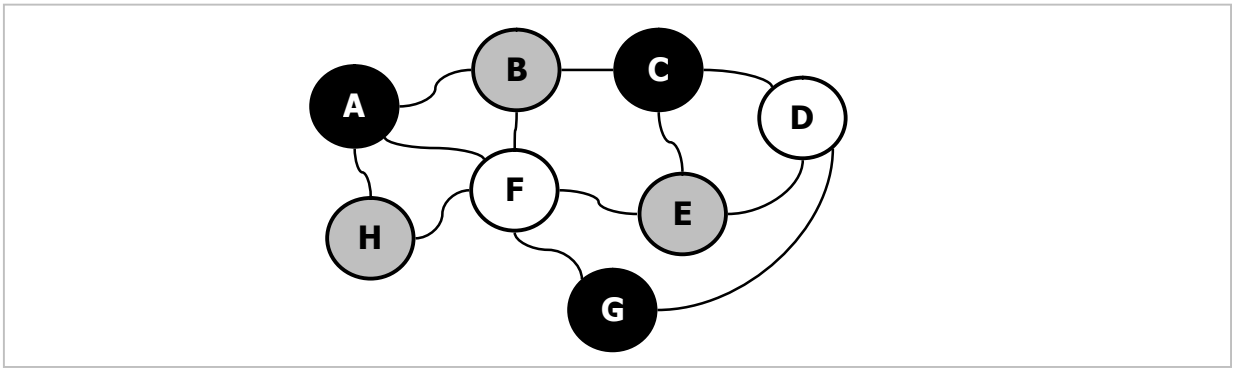


Рис. 19-1 — Трёхцветная раскраска графа

Суть, разумеется, не в красоте, — вот случай из жизни. Надо перевезти по тряской дороге расфасованные вещества, способные вступать между собой в опасные химические реакции. Тогда уложим их в контейнеры так, чтобы при нарушении упаковки лежащие вместе вещества не могли вступить в реакцию. Ёмкость контейнера не ограничена, но количество их должно быть минимально.

Сведём это к задаче раскраски. Представим каждое вещество вершиной графа и соединим рёбрами взаимно реагирующие вещества. Тогда соответствующие инцидентным вершинам вещества нельзя класть в один контейнер. Пусть все контейнеры покрашены в разные цвета, тогда можно вообразить покрашенными в те же цвета и помещённые в них упаковки с веществами.

В задаче раскраски ставятся две цели: a) определить минимально необходимое количество контейнеров (красок) и b) найти распределение упаковок по контейнерам, то есть раскраску вершин конкретного графа.

19.2. Точное решение

Вспомним о множествах независимых вершин (глава 17). В каждом таком множестве (рис. 19-2) все вершины попарно не инцидентны, и могут быть окрашены одним цветом — уложены в один контейнер. Но как быть с оставшимися вершинами?

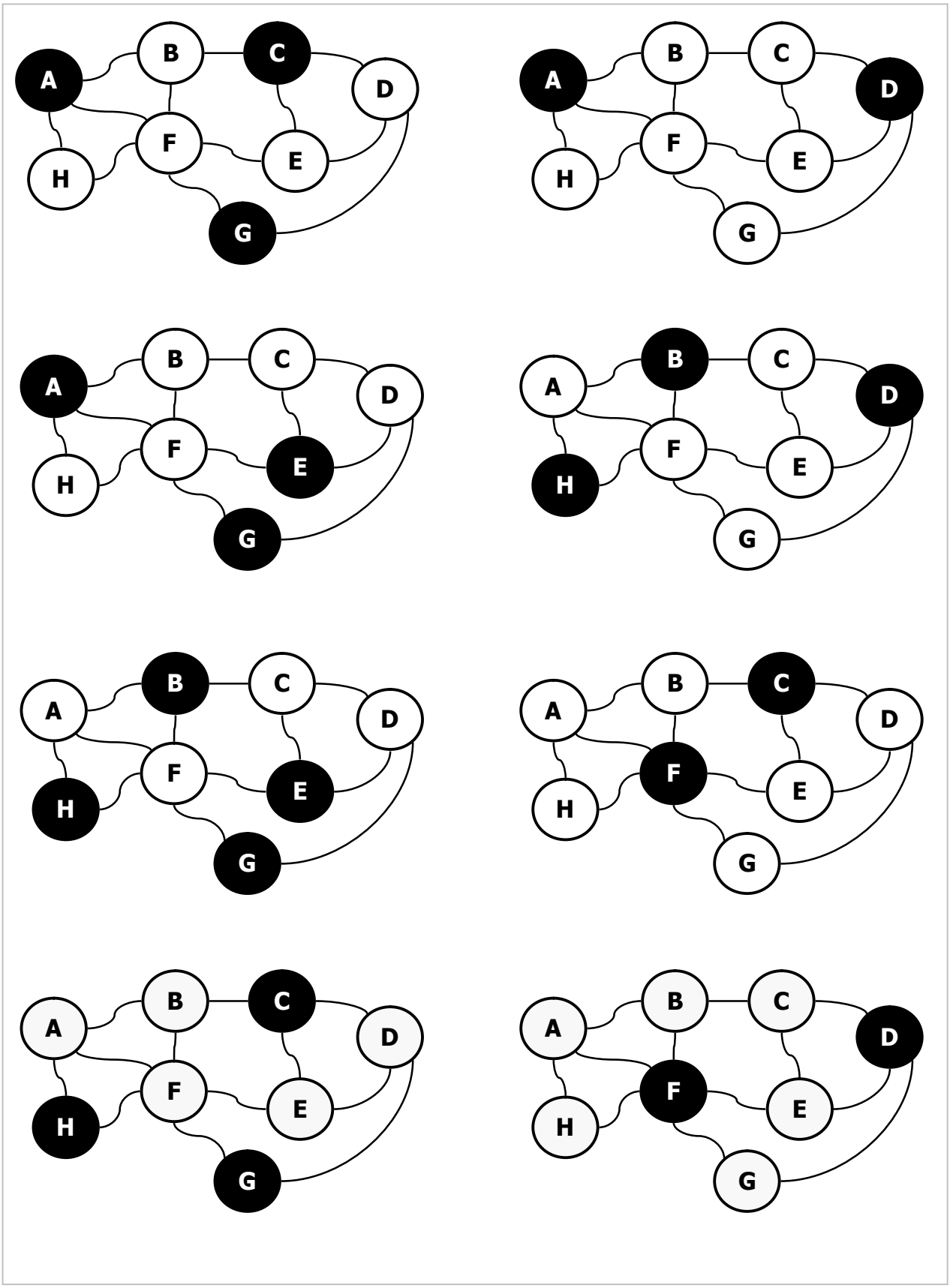


Рис. 19-2 — Все подмножества независимых вершин графа

Не будем торопиться и вначале решим менее сложную задачу: определим минимальное количество требуемых красок — *хроматическое число* графа (минимальное число контейнеров). Для этого найдём и поместим их в табл. 19-1 все максимальные независимые множества вершин графа.

Табл. 19-1 — Все независимые множества вершин

Условный цвет	Независимые подмножества вершин							
1	A	-	C	-	-	-	G	-
2	A	-	-	D	-	-	-	-
3	A	-	-	-	E	-	G	-
4	-	B	-	D	-	-	-	H
5	-	B	-	-	E	-	G	H
6	-	-	C	-	-	F	-	-
7	-	-	C	-	-	-	G	H
8	-	-	-	D	-	F	-	-
Сумма подмножеств	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Каждое из этих подмножеств вершин можно без опаски покрасить своим цветом, условно обозначенным числом в левой колонке. Если найти минимальный набор подмножеств, полностью покрывающий граф, то количество этих подмножеств и будет хроматическим числом. Здесь для этого подходят три подмножества с «цветами» **1**, **5** и **8**: **A–C–G**, **B–E–G–H** и **D–F**, они выделены серым. Значит, хроматическое число графа составляет 3. Понятно, что такой минимальный набор подмножеств ищется как минимальное покрытие (ЗНП), — этим же приёмом мы пользовались для поиска доминирующих вершин в главе 18.

Отсюда рукой подать до розыска одной из оптимальных раскрасок графа (здесь тремя красками). Если бы найденные подмножества составили *разбиение*, то стали бы решением задачи. Напомню, что в разбиении подмножества взаимно *не пересекаются*. Однако здесь вершина *G* входит в два подмножества и претендует на запрещённую двойную окраску. Но если оставить её только в одном из подмножеств, проблема исчезнет, — далее мы так и поступим, превратив *покрытие* в *разбиение*.

19.3. Приближённое решение

Итак, мы пришли к тому, что путь к точному решению задачи о раскраске лежит через решение двух промежуточных задач:

- формирование всех множеств независимых вершин:
- решение задачи о наименьшем покрытии (ЗНП).

Вторая задача, как известно, NP-сложна, и даже для умеренного объёма данных может затребовать неприемлемое время. Но есть несколько быстрых и «почти оптимальных» алгоритмов раскраски. «Почти» потому, что количество истраченных при этом красок иногда может превышать хроматическое число. Один из таких приёмов — быстрое градиентно» решение задачи о наименьшем покрытии, найденное нами в главе 11. Как правило, оно даёт результат не более чем на 25% хуже оптимального.

19.4. Листинг

Алгоритм поиска одной из оптимальных раскрасок реализован в методе **`TGraph.GenPaints`**, он генерирует множество подмножеств вершин, каждое из которых может быть окрашено своим цветом. Отмечу, что поле **`TNode.mColor`** здесь не затрагивается, и никакой реальной окраски вершин не происходит.

Листинг 19-1 — Генерация оптимальной или субоптимальной раскраски

```
function TGraph.GenPaints(aQuick: boolean): TSet;
var  Undep : TSet;        // множество независимых множеств
       Buf : TBuffer;  // рабочий буфер
         S : TSet;        // текущее подмножество вершин
        CS : TCostSet;    // оценённое подмножество вершин
     Cover : TCostSet;    // минимальное покрытие
begin
  Undep:= GenUndepend;    // генерируем все независимые подмножества вершин
  // Переносим независимые множества в буфер оценённых подмножеств
  // для последующего поиска минимального покрытия
  Buf:= TBuffer.Create;       // создаём рабочий буфер
  S:= Undep.GetFirst as TSet;  // первое независимое множество
  while Assigned(S) do begin   // перебор независимых множеств
    // Создаём оценённые подмножества (цена Cost=1)
    // и помещаем в буфер:
    CS:= TCostSet.Create(1, S.Copy as TSet, true);
    Buf.Push(CS);
    S:= Undep.GetNext as TSet;
  end;
  // Уничтожаем множество максимальных независимых множеств
  Undep.ClrAndDestroy;
  Undep.Free;
  // Формируем минимальное покрытие вершин (unit Assembly):
  if aQuick
    then Cover:= CollectGradCover(mNodes, Buf)  // быстро
    else Cover:= CollectMinCover(mNodes, Buf);  // точно
  RemoveItemsFromBuf(Buf, Cover.mSet);  // удаляем результат из буфера
  Buf.ClrAndDestroy;                    // а буфер очищаем
  Buf.Free;                             // и удаляем
  // Формируем результат:
  Result:= Cover.mSet.Copy as TSet;  // копия множества-покрытия
  Cover.Free;                        // удаляем покрытие
  Result.CoverToDissect;             // превращаем покрытие в разбиение
end;
```

Метод принимает булев параметр, влияющий на ускорение при поиске наименьшего покрытия: **`false`** — требуется точное решение; **`true`** — необходимо быстрое решение (но количество красок может превысить хроматическое число).

После генерации множества всех максимальных независимых вершин (в переменной **Undep**) формируем буфер оценённых подмножеств **Buf**, и далее «скармливаем» его функциям **CollectMinCover** или **CollectGradCover**. Найденные так покрытия копируем в результат, удаляем из временного буфера, а буфер уничтожаем. В последней строке вызываем специфический метод **TSet.CoverToDissect**, превращающий наименьшее покрытие в одно из наименьших разбиений (листинг 19-2).

Листинг 19-2 — Преобразование наименьшего покрытия в разбиение

```
// Преобразование покрытия в разбиение
// Элементами множества должны быть TCostSet!

procedure TSet.CoverToDissect;
var   i, j : integer;  // индексы
    Si, Sj : TCostSet; // текущие элементы
begin
  for i:= 1 to GetCount-1 do begin
    Si:= GetItem(i) as TCostSet;
    // Вычитаем из текущего подмножества все последующие
    for j:= i+1 to GetCount do begin
      Sj:= GetItem(j) as TCostSet;
      Si.mSet.Sub(Sj.mSet);          // Si:= Si - Sj
    end;
  end;
end;
```

19.5. Испытание

Ниже показана программа для испытания раскрасок графа:

Листинг 19-3 — Программа для испытания раскрасок

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    Paints: TSet;
    S : string;

begin
  repeat
    Gr:= TGraphChars.GenRandom(false, 1, 1, 20, 50);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    Paints:= Gr.GenPaints(true);  // быстро
    Paints.Expo;
    Paints.ClrAndDestroy;
    Paints.Free;
```

```
    Writeln('- - - - - - - -');
    Paints:= Gr.GenPaints(false); // точно
    Paints.Expo;
    Paints.ClrAndDestroy;
    Paints.Free;
    Writeln('- - - - - - - - - - - - - - - - - -');
    Write('Quit ? ');
    Readln(S);
    Gr.Free; // Освобождение графа
  until S<>'';
end.
```

19.6. Итоги

19.6.1. Суть задачи раскраски: окрасить вершины графа минимальным набором красок так, чтобы вершины в любых смежных парах оказались разноцветными.

19.6.2. Минимальное количество красок, необходимое для раскраски, называется хроматическим числом графа.

19.6.3. В ходе поиска оптимальной раскраски решаются две промежуточные задачи: a) формирование всех независимых подмножеств вершин и b) поиск наименьшего покрытия с последующим преобразованием его в разбиение.

19.6.4. Точное решение задачи о раскраске экспоненциально сложно, поэтому иногда прибегают к менее точным, но быстрым методам.

19.7. Задания

19.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	стр. 75
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 20
Центры графа

Вновь обратимся к задачам оптимального размещения служб и рассмотрим в этой связи несколько примеров.

20.1. Центры: внешние, внутренние и внешне-внутренние

Пусть мосты с островами образуют связанный граф, либо сильно связанный орграф. Мы помним, что в таких графах любая вершина достижима из любой другой, хотя в орграфе прямой и обратный пути между вершинами отличаются. Найдём место для базирования единственной пожарной части. Разумно будет найти такой остров, путь от которого до самого удалённого от него острова был бы минимален. Задачи такого рода называют *минимаксными*. Из этих же соображений выбирают и место размещения скорой помощи, работающей по вызову. Вершину, или подмножество вершин, отвечающих этому требованию, называют *внешним центром* графа.

Для графа на рис. 20-1 внешним центром является вершина *A*, путь из которой к любой другой вершине (внешний радиус **ρ**) не превышает 3-х единиц.

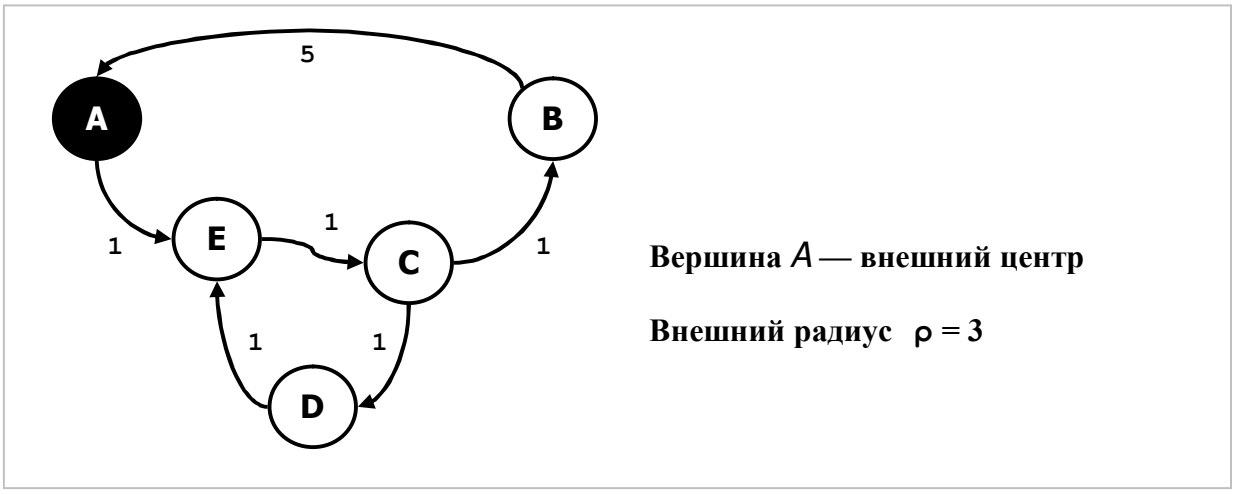


Рис. 20-1 — Внешний центр орграфа

А где место для медпункта, к которому пациент добирается сам? Здесь надо минимизировать расстояние от самого удалённого острова к центру, такой центр называют *внутренним*. Для графа на рис. 20-2 внутренним центром является вершина *B*, расстояние к которой из любой другой вершины (внутренний радиус **ρ**) не превышает 3-х единиц.

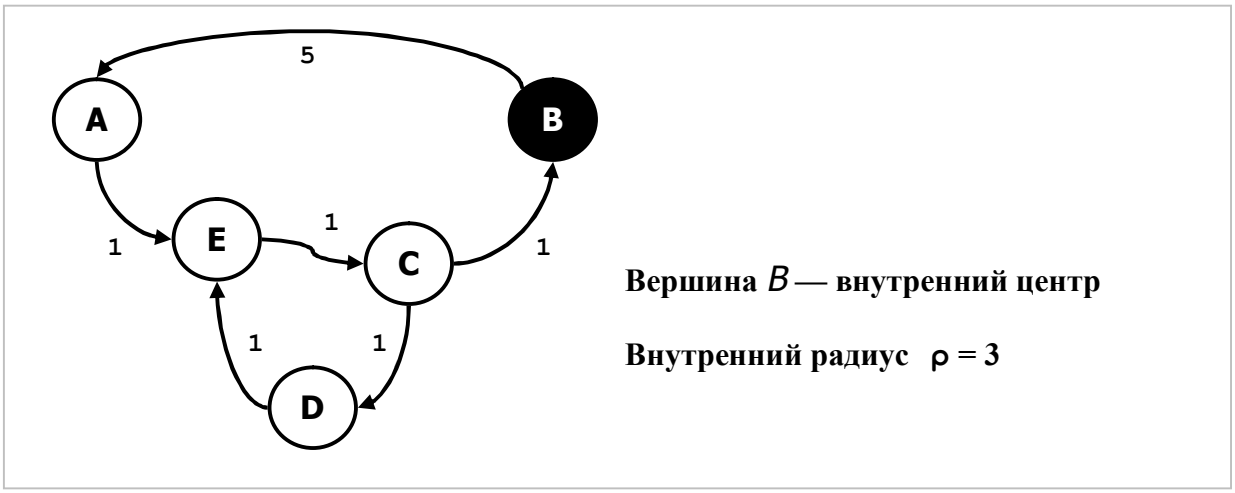


Рис. 20-2 — Внутренний центр орграфа

Наконец, подумаем о размещении больницы вкупе со скорой помощью, когда пациента доставляют в больницу на карете. Тогда следует минимизировать общее расстояние от больницы до пациента и обратно, такой центр называют *внешне-внутренним*.

Понятно, что в *неориентированном* графе все три упомянутых центра совпадают. Центры неориентированного графа, показанного на рис. 20-3, представлены множеством вершин *C-D-E* : здесь расстояние между любой из них и другими вершинами графа (радиус ρ) не превышает 2-х единиц. Тут экстренную службу можно разместить в любой из трёх вершин центра (но не во всех сразу).

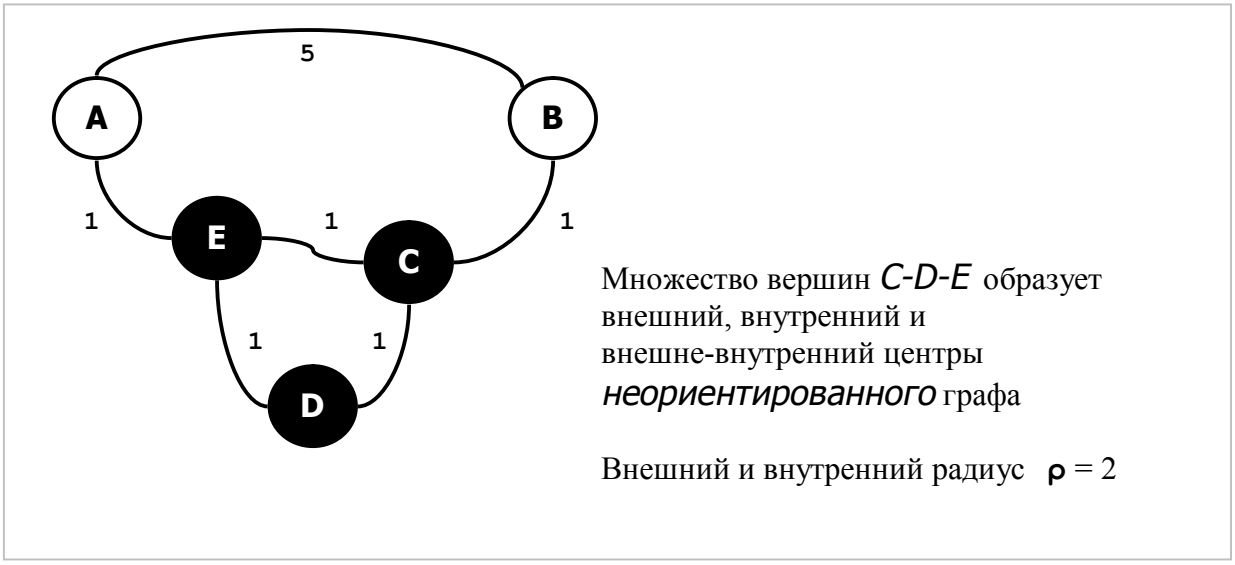


Рис. 20-3 — Центр неориентированного графа

Далее мы построим алгоритмы для поиска *внутренних*, *внешних* и *внешне-внутренних* центров графов и орграфов.

20.2. Взвешенное расстояние

В ходе работы нам потребуется расширенное понятие расстояния — *взвешенное расстояние*. Его определяют как произведение расстояния между вершинами на вес одной из вершин. Нечто подобное существует в механике: это момент силы, определяемый как произведение силы **F** на плечо **L**. При равенстве моментов рычаг сохраняет равновесие (рис. 20-4).

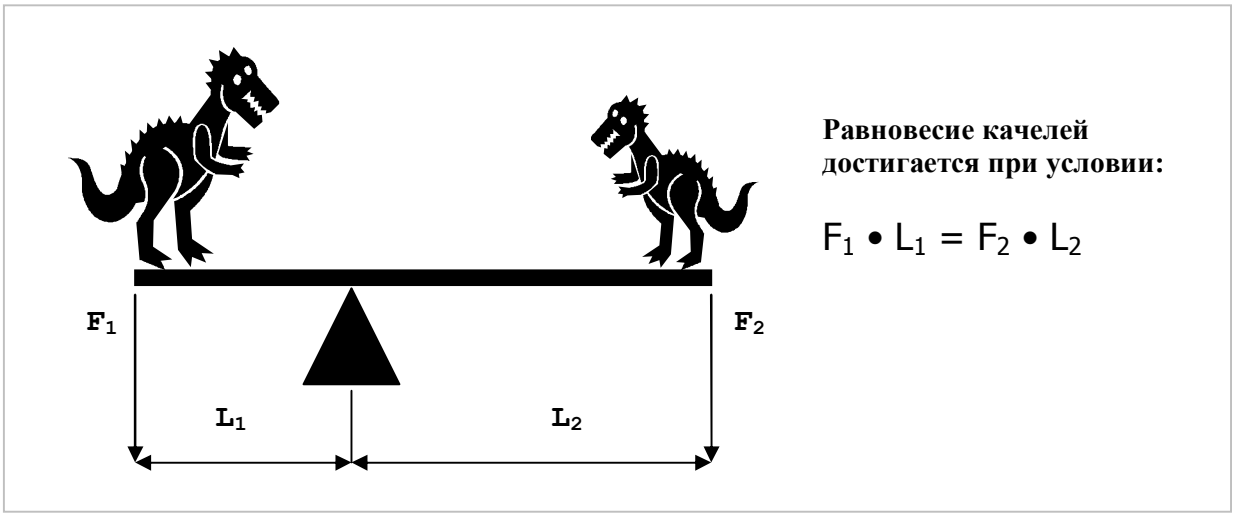


Рис. 20-4 — Пример равенства взвешенных расстояний (моментов)

Вот пример, где выручает взвешенное расстояние. Пусть почтовая связь между островами налажена через центр, куда почта стекается со всех других островов, сортируется и вновь развозится по островам. Объём почты для каждого острова пропорционален его населённости (весу вершины графа). Положим, что почта доставляется самолётами, в которых часть полезного груза можно замещать дополнительным топливом, и тогда дальность полёта возрастает с уменьшением полезной нагрузки. Оттого самолёт может доставить, либо большой груз на малое расстояние, либо малый груз — на большое. Тогда произведение дальности самолёта на его грузоподъёмность можно грубо оценить константой **λ** — *взвешенной дальностью*. Какова должна быть эта константа **λ**, характеризующая самолёты, и где поместить центр сортировки почты? Очевидно, что **λ** должна быть по возможности меньше (лёгкие самолёты дешевле), а почтовый центр надо расположить так, чтобы все острова оказались достижимы в пределах этой минимальной *взвешенной* дальности.

На рис. 20-5 почтовый центр размещен в вершине *C*. Наибольшие взвешенные расстояния пролегают от него к вершинам *D* и *E* и составляют **2•4=8** условных единиц, — именно таким должно быть оптимальное произведение дальности на грузоподъёмность самолёта.

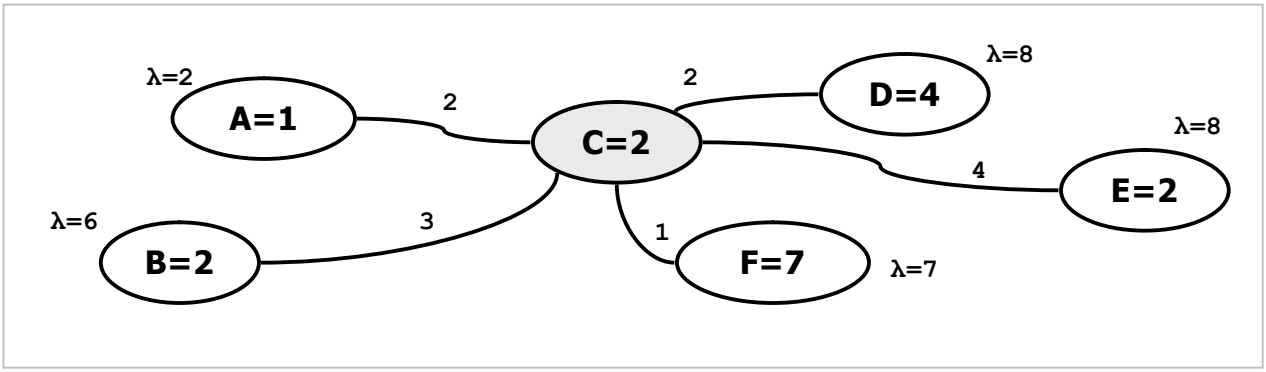


Рис. 20-5 — Центр сортировки почты в вершине *C*

Отметим, что в отличие от обычного расстояния, взвешенное расстояние *не симметрично* (даже в неориентированном графе). Например, взвешенное расстояние $A \to C$ составляет **2•2=4** единицы, а обратное $C \to A$ равно **1•2=2** единицы.

В поисках центра, мы будем оперировать с взвешенными расстояниями. В графах, где вершины не взвешены, они совпадают с обычными. И, поскольку примеры на таких графах наглядней, я буду прибегать к ним.

20.3. Разделения и радиусы

Рассмотрим орграф с единичными (для простоты) весами вершин (рис. 20-6). Возьмём вершину *D*, и найдём наиболее удалённую от неё по направлению стрелок — это вершина *B*, расстояние до которой составляет **2+4+1+3+1=11** единиц. Если бы вес вершины *B* составлял, например, **5**, то взвешенное расстояние к ней составило бы **11•5=55**. Взвешенное расстояние к самой удалённой вершине по направлению стрелок называют *внешним числом разделения* для данной вершины. Итак, внешнее число разделения для вершины *D* составляет 11.

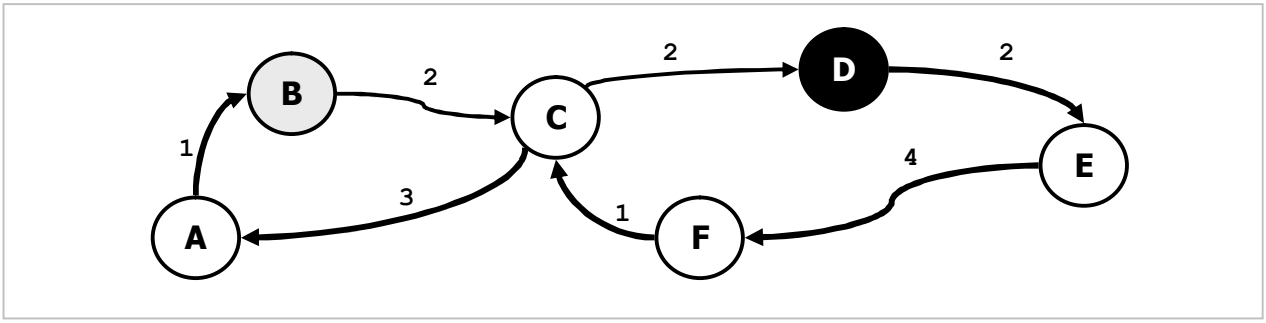


Рис. 20-6 — Внешнее число разделения для вершины *D* равно 11 (расстояние до *B*)

Определив внешние числа разделения для всех вершин, можно выбрать вершину (или несколько), число разделения которой будет минимально, — она

(или они) составят *внешний центр* графа. Минимальное внешнее число разделения называют *внешним радиусом*.

Теперь поступим наоборот, и найдём вершину, *из которой* путь к выбранной нами вершине *D* максимален, это будет вершина *E* (рис. 20-7). Соответствующее расстояние — *внутреннее число разделения* — равно **2+1+4=7**. Заметим, что если бы вес вершины *E* составлял, к примеру, **5**, то внутреннее число разделения для вершины *D* составило бы **7•5=35**.

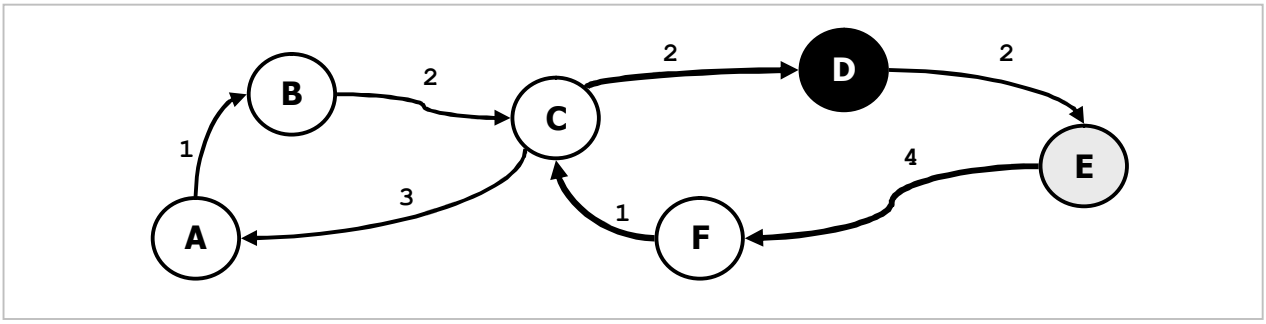


Рис. 20-7 — Внутреннее разделение для вершины *D* равно 7 (расстояние от *E*)

Найдя внутренние числа разделения для всех вершин, мы можем отобрать одну или несколько вершин с минимальным внутренним числом разделения — *внутренний центр* графа. Соответствующее минимальное внутреннее число разделения будет *внутренним радиусом*.

Очевидно, что для неориентированного графа *внешние* и *внутренние* числа разделения, и соответствующие центры с радиусами совпадают.

20.4. Алгоритм поиска центров

Предыдущие рассуждения ведут к той мысли, что для поиска центра надо сначала найти обычные расстояния между всеми вершинами. По ним легко определить *взвешенные* расстояния от любой вершины ко всем остальным путём умножения расстояний на вес целевых вершин. Останется лишь дважды перебрать вершины: сначала в поиске минимального числа разделения, а затем в розысках множества вершин, которое ему удовлетворяет.

Некоторую трудность доставляет лишь первая стадия — вычисление всех расстояний (не взвешенных) между вершинами графа. Однако схожая проблема уже решена нами в главе 15 методом Флойда-Уоршелла через составление карты дальних указателей. Здесь поступим схожим образом, и даже воспользуемся теми же элементами данных: дальними указателями **TFarLink**, хранящими в числе прочего расстояние к целевой вершине, а также полем **TNode.mFarLinks** — списком таких указателей. Однако новая процедура составления карты указателей будет снабжена параметром, указывающим её назначение: поиск внутреннего, внешнего или внешне-внутреннего центра. Для неориентированного графа этот

параметр может быть любым и на результат не влияет. Тип параметра определён в модуле **Graph** как перечисление:

```
TCenter = (   Undefined,  // направление карты не определено
              InCenter,   // внешний центр и медиана
              OutCenter,  // внутренний центр и медиана
              InOutCenter // внешне-внутренний центр и медиана
          );
```

Итак, для поиска центров создадим основной и два вспомогательных метода:

- **TGraph.GenCenter** — основной метод поиска центра;
- **TGraph.InitMap** — вспомогательный метод построение карты;
- **TGraph.DoneMap** — вспомогательный **м**етод уничтожения карты;

Начнём со вспомогательных методов, выполняющих основную работу.

Метод **InitMap** строит распределённую по вершинам графа карту дальних указателей:

Листинг 20-1 — Вспомогательный метод инициализации карты дальних указателей

```
procedure TGraph.InitMap(aDirect: TCenter);

  // aDirect = InCenter | OutCenter | InOutCenter — направление поиска

  //- - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode; aDirect : TCenter);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
    //- - - - - - - - - - - - - - - - - - - - -
    // Для поиска внутреннего центра орграфа или центра графа
    procedure LocalIn;
    begin
      with Node do begin
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = aNode then begin  // сравнение с конечной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - -
    // Для поиска внешнего центра орграфа
    procedure LocalOut;
    begin
      with Node do begin
        Link:= InLinkFirst;             // первая входящая связь
```

```
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mOwner = aNode then begin // сравнение с исходной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= InLinkNext;     // следующая исходящая связь
        end; // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - -
  begin   { InitFarLinks }
    PosPush;  // сохранить позицию перебора
    Node:= NodeFirst;                // перебор всех вершин графа
    while Assigned(Node) do begin    // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // это связь на себя
      end else with Node do begin
        // Создать связь на очередную вершину
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        if (aDirect=InCenter) or not mDirect
          then LocalIn   // Внутренний центр орграфа или центр графа
          else LocalOut; // Внешний центр орграфа
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;  // перебор всех вершин графа
    end;
    PosPop;  // восстановить позицию перебора
  end;
  //- - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
         FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;  // указатель из aNode
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
```

```
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
                   then Result:= true;      // признак того, что изменён
          FL2:= GetNext as TFarLink;        // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
  procedure Create(aDirect : TCenter);
  var  Node : TNode;    // текущая вершина
       Step : integer;  // этап обработки (номер цикла)
       Flag : boolean;  // признак продолжения обработки
  begin
    // Предварительная очистка карты:
    Node:= NodeFirst;
    while Assigned(Node) do begin          // перебор вершин
      with Node do if Assigned(mFarLinks) // если карта существует
        then mFarLinks.ClrAndDestroy      // очищаем
        else mFarLinks:= CreateSet;       // иначе создаём пустую
      InitFarLinks(Node, aDirect);        // инициализация дальних указателей
      Node:= NodeNext;
    end;

    // Обработка вершин

    for Step:= 1 to mNodes.GetCount-2  do begin
      Flag:= false;
      Node:= NodeFirst;
      while Assigned(Node) do begin // перебор вершин
        if Handle(Node, Step)       // если обновлялись дальние указатели,
          then Flag:= true;         // то отметить это установкой флага
        Node:= NodeNext;
      end;
      // если дальние указатели не обновлялись, то прервать цикл
      if not Flag then Break;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -

var  Node : TNode;      // текущая вершина
     Buf : TBuffer;  // буфер для хранения списков дальних указателей
  //- - - - - - - - - - - - - - - - - - - - -
  // Сохранение в буфере текущих расстояний
  procedure SaveInBuf;
  begin
    Node:= NodeFirst;
    // Сохранение списков в буфере:
    while Assigned(Node) do begin
      Buf.Put(Node.mFarLinks);
      Node.mFarLinks:= nil;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
```

```
  // Извлечение из буфера и добавление расстояний
  procedure AddFromBuf;
  var     S : TSet;       // список из буфера
        FL1 : TFarLink;   // дальний указатель из S
        FL2 : TFarLink;   // дальний указатель из Node
  begin
    Node:= NodeFirst;
    // Перебор вершин:
    while Assigned(Node) do begin
      S:= Buf.Get as TSet;
      FL1:= S.GetFirst as TFarLink;
      while Assigned(FL1) do begin
        FL2:= Node.GetFarLink(FL1.mNodeFar);
        if Assigned(FL2) then FL2.mDist:= FL2.mDist + FL1.mDist;
        FL1:= S.GetNext as TFarLink;
      end;
      S.ClrAndDestroy;
      S.Free;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - -
begin  { TGraph.InitMap }

  // Выход, если карта уже создана
  if mMapDirect = aDirect  then Exit;

  mFlagMap:= aDirect; // запоминаем направление карты

  case aDirect of
    InCenter,   // внешний центр и медиана
    OutCenter:  // внутренний центр и медиана
      Create(aDirect);
    InOutCenter: // внешне-внутренний центр и медиана
      begin
        // Строим карту для входящих гамм:
        Create(InCenter);
        // Создаём буфер и сохраняем в нём расстояния между вершинами:
        Buf:= TBuffer.Create;
        SaveInBuf;
        // Строим карту для исходящих гамм:
        Create(OutCenter);
        // Прибавляем расстояния, сохранённые в буфере,
        // и освобождаем буфер:
        AddFromBuf;
        Buf.ClrAndDestroy;
        Buf.Free;
      end;
  end;
  // Сортируем списки дальних указателей по неубыванию расстояния
  // (необходимо при поиске медиан)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.SortFarLinks;
    Node:= NodeNext;
  end;

end;
```

Метод включает ряд внутренних локальных процедур. Он создаёт распределённую по вершинам графа карту дальних указателей между парами

вершин. Обратим внимание на построение внешне-внутренней карты. Здесь последовательно строятся две карты: сначала внутренняя, затем внешняя, после чего прямые и обратные расстояния к вершинам суммируются. Для сохранения данных внутренней карты, перед построением внешней карты они переносятся во временный буфер.

Метод **DoneMap** уничтожает ранее созданную карту:

```
// Освобождение карты дальних указателей

procedure TGraph.DoneMap;
var  Node : TNode;     // текущая вершина
begin
  // Если очищена, то выход:
  if mMapDirect = Undefined then Exit;
  // Очистка распределённой по вершинам карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      if Assigned(mFarLinks) then begin
        mFarLinks.ClrAndDestroy;    // уничтожаем указатели
        mFarLinks.Free;             // и список
        mFarLinks:= nil;
      end;
    end;
    Node:= NodeNext;
  end;
  mMapDirect:= Undefined;  // признак очищенной карты
end;
```

Основной метод, пользуясь картой, формирует центр графа (множество вершин), а также возвращает взвешенный радиус графа **aLambda**.

Листинг 20-2 — Функция формирования центра графа

```
// Возвращает множество вершин центра и взвешенный радиус aLambda
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenCenter(var aLambda: integer; aDirect: TCenter): TSet;

  //- - - - - - - - - - - - - - - - - - - -
  // Поиск максимального взвешенного расстояния от вершины aNode.
  // Использует ранее построенную карту дальних указателей mFarLinks
  function GetMaxDist(aNode: TNode): integer;
  var FL: TFarLink;
      Value: integer;
  begin
    Result:= 0;
    with aNode.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if mLoadNodes   // если вершины графа нагружены...
          then Value:= FL.mNodeFar.mValue   // для нагруж. вершин
          else Value:= 1;                   // если вершины не нагружены
        if Result < FL.mDist * Value
          then Result:= FL.mDist * Value;
        FL:= GetNext as TFarLink;
      end;
```

```
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - -
var Dist, MinDist: integer;  // текущее и минимальное взвеш. расстояние
    Node: TNode;

begin  { TGraph.GenCenter }
  // Предварительно формируем карту дальних указателей,
  // в которой каждый элемент содержит расстояние между парами вершин.
  DoneMapAndSet;     // если карта создана, то ликвидируем её
  InitMap(aDirect);  // строим карту с заданным направлением

  // Выбор минимального из максимальных взвешенных расстояний
  MinDist:= MaxInt;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);
    if MinDist > Dist then MinDist:= Dist;
    Node:= NodeNext;
  end;
  Result:= CreateSet;
  // Формирование множества-центра
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);    // число разделения очередной вершины
    if Dist = MinDist then Result.Insert(Node);
    Node:= NodeNext;
  end;
  aLambda:= MinDist;  // минимальное из максимальных расстояний
end;
```

20.5. Испытание

Следующая программа отыскивает все центры графа.

Листинг 20-3 — Программа поиска центров

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    Center: TSet;
    Lambda : integer;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;

  Writeln('- - - - - - - OUT CENTER - - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, OutCenter);
```

```
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;

  Writeln('- - - - - - - IN CENTER - - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, InCenter);
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;
  Writeln('- - - - - - IN-OUT CENTER - - - - - - - - - -');
  Center:= Gr.GenCenter(Lambda, InOutCenter);
  Center.Expo;
  Writeln('Lambda= ', Lambda);
  Center.Free;
  Gr.Free; // Освобождение графа
  Readln;
end.
```

Испытание на показанном ниже взвешенном орграфе (см. также рис. 20-8):

```
Орграф с взвешенными дугами и вершинами
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

дало результат, представленный в таблице:

Центр	Вершины центра	Взвешенный радиус λ
Внешний	**G**	**126**
Внутренний	**A**	**128**
Внешне-внутренний	**A, B, E, G, H**	**296**

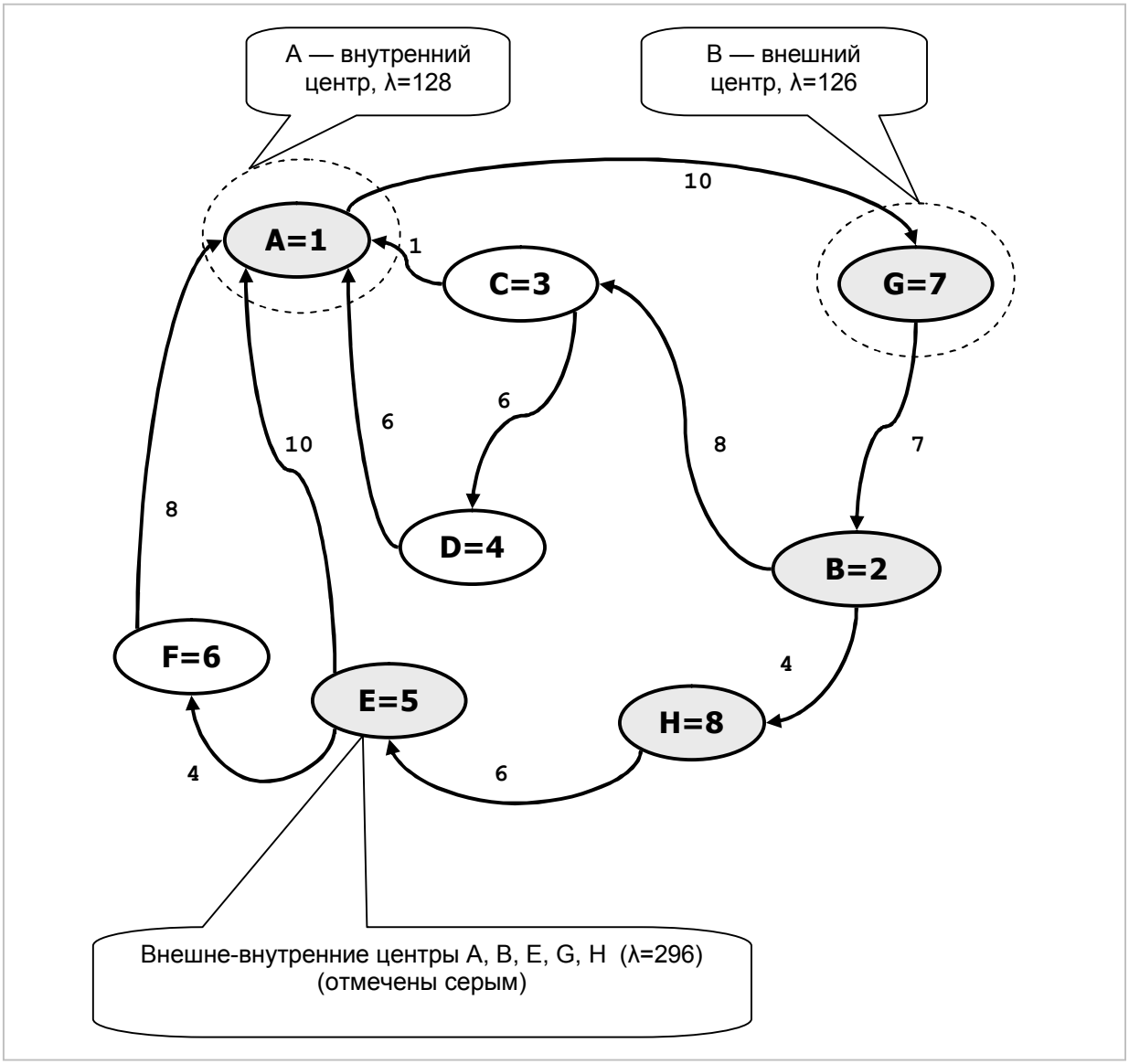


Рис. 20-8 — Центры орграфа с взвешенными дугами и вершинами

На роль внешне-внутреннего центра претендуют пять вершин, — любая из них годна для этого. Эти вершины образуют цикл, а взвешенный радиус этого внешне-внутреннего центра равен длине цикла, умноженному на вес самой «тяжёлой» вершины в нём:

(10+7+4+6+10)•8 = 296.

Испытание на похожем неориентированном графе (рис. 20-9):

```
Неориентированный граф с взвешенными дугами и вершинами
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

дало такой результат:

Центр	Вершины центра	Взвешенный радиус λ
Внешний	**H**	**77**
Внутренний	**H**	**77**
Внешне-внутренний	**H**	**154**

В неориентированном графе все три центра совпадают, а внешне-внутренний радиус, естественно, вдвое больше внутреннего и внешнего.

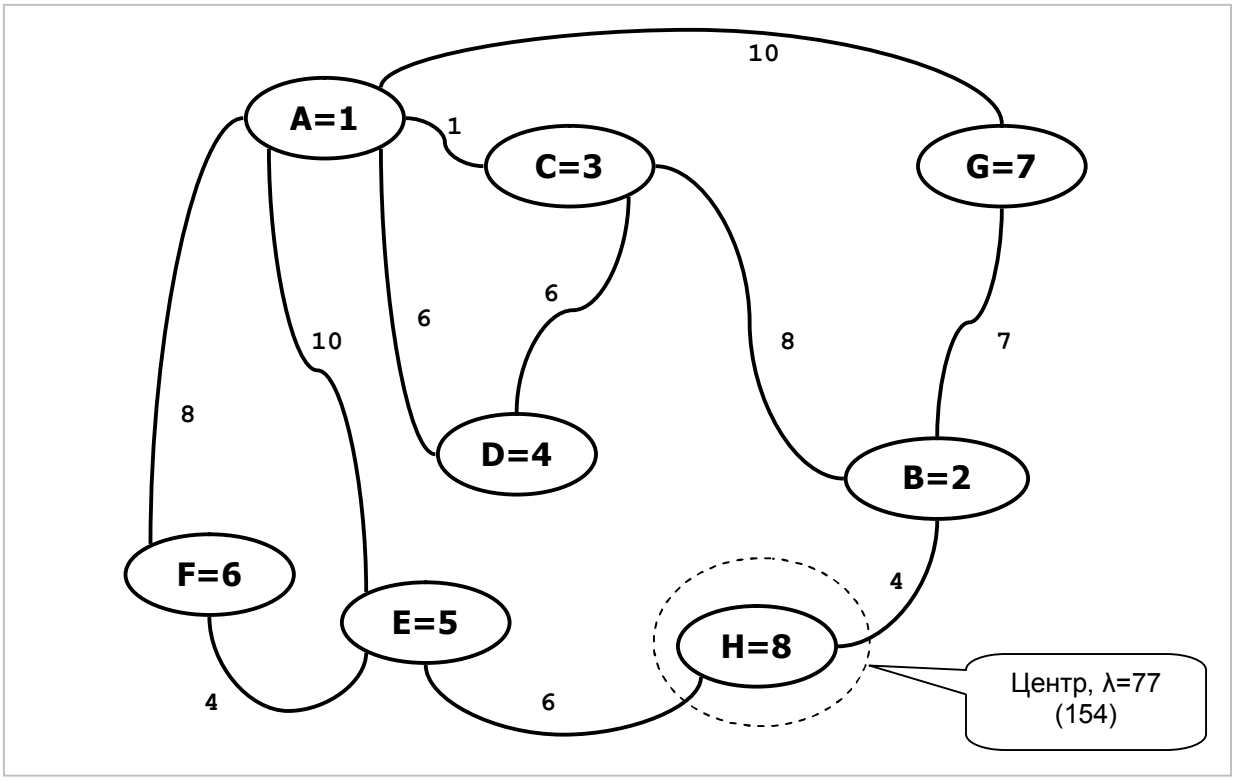


Рис. 20-9 — Центр неориентированного графа с взвешенными дугами и вершинами

20.6. Итоги

20.6.1. Задачи поиска внешних, внутренних и внешне-внутренних центров связывают с размещением экстренных служб, хотя этой сферой модель не ограничена.

20.6.2. При поиске центров учитывается как длина дуг (рёбер), так и веса вершин, для чего вводится понятие взвешенного расстояния. Если вершины графа не взвешены или имеют одинаковый вес, то взвешенное расстояние становится эквивалентным обычному.

20.6.3. Основная работа при поиске центров состоит в построении карты дальних указателей методом Флойда-Уоршелла.

20.7. Задания

20.7.1. Внешне-внутренний центр

Постройте метод поиска внешне-внутреннего центра ориентированного графа.

20.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 21
P-центры

21.1. Что такое p-центры?

В предыдущей главе найден *центр* графа — место для *единственной* экстренной службы (пожарной части, скорой помощи и т.п.). Службу размещают в одной из вершин графа — центре — так, чтобы взвешенное расстояние от неё до самой удалённой вершины оказалось минимальным.

А если позволительны несколько пожарных команд, например, две или три? Собирать их в одном месте явно неразумно. Очевидно, что равномерное распределение их по графу сократит расстояние до самой дальней точки. Задачи этого рода называют задачами о размещении *p-центров*, где *p* – это количество центров. Поли-центры (от греч. πολύς — «многочисленный»), так же как и центры, могут быть *внешними*, *внутренними* и *внешне-внутренними*.

Существуют два рода задач о p-центрах: *прямая* и *обратная*.

В *прямой задаче* задан предельный взвешенный радиус **λ** и требуется найти множество удовлетворяющих ему центров (каждый центр может включать одну или несколько вершин).

В *обратной задаче* известно *количество* центров, а требуется найти их распределение по вершинам и подобающий минимальный взвешенный радиус **λ.**

Поскольку обратная задача легко решается через прямую, основное внимание уделим прямой задаче.

21.2. Геометрическая разминка

Идею решения прямой задачи рассмотрим на геометрической аналогии. Пусть точки на плоскости (рис. 21-1) — это острова в океане, а число **R** — предельная дальность тамошних самолётов. Будем искать минимальное подмножество точек, подходящих для промежуточных аэродромов. Таковыми будут те из них, из которых остальные точки достижимы в пределах радиуса **R**.

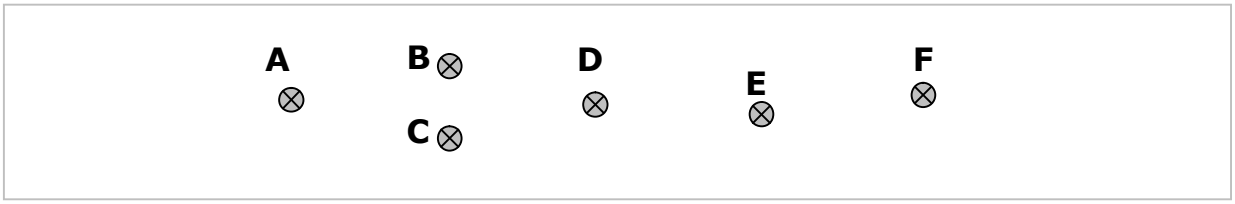


Рис. 21-1 — Ряд точек на плоскости

Из каждой точки очертим круг радиуса **R** (на рис. 21-2 показаны лишь три из шести кругов). Понятно, что центр круга достижим из всех лежащих в круге точек,

включая и сам центр. Назовём точки в круге *ограниченной входящей гаммой* по отношению к центру круга. Например, входящая гамма для *D* — это точки *B, C, D, E*.

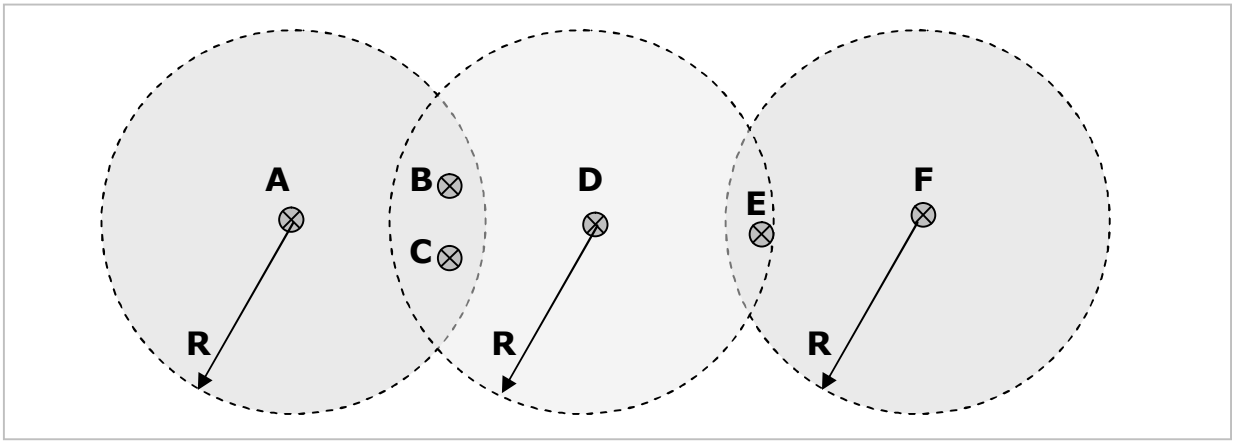


Рис. 21-2 — Круги, соответствующие центральным точкам A, D, F

Далее зададимся вопросом: если некоторая *центральная* точка не попадает в круг другой *центральной* точки, то из каких точек достижимы они обе? Ясно, что из тех точек, что лежат в пересечении входящих гамм (на пересечении кругов). Так, центральные точки *A* и *D* достижимы обе только из точек *B* и *C*. Аналогично пара точек *D* и *F* достижима только из точки *E* . А пара точек *A* и *F* вообще не достижима из других точек, поскольку их круги (гаммы) не пересекаются. Очевидно, что точки, лежащие на *пересечении* кругов, с большим правом претендуют на роль центров, поскольку из них достижимо больше других точек.

Найдя все области пересечений, из которых достижимы *пары* точек (центры кругов), мы вправе заинтересоваться областями, из которых достижимы *тройки* точек. Ясно, что к паре точек можно добавить третью, если пересечение её гаммы с гаммой этой пары не пусто. Можно продолжать процесс, находя области, из которых достижимы *четвёрки*, *пятёрки* точек и так далее, пока дальнейшее расширение множества достигаемых точек станет невозможным (пересечения гамм окажутся пустыми). Так, если не обнаружится областей, из которых достижимы *четвёрки* точек, то нет смысла искать области для *пятёрок*.

В ходе накопления всё новых областей можно отбрасывать те подмножества достигаемых точек, которые входят в более мощные. Например, если какая-то пара точек входит ещё и в тройку, то пару отбрасываем. В итоге получим сравнительно небольшой набор из множеств центральных точек и соответствующих им *пересечений* — кандидатов на роль p-центров. Затем алгоритм поиска наименьшего покрытия (ЗНП), найдёт минимальный набор множеств достигаемых точек, в сумме покрывающий все точки. Соответствующие этим множествам *области пересечений* и будут искомыми p-центрами.

На рис. 21-3 показан результат поиска точек, из которых все остальные достижимы в пределах радиуса **R**. Здесь найдены два центра. В первый входят две

точки (*B* и *C*) — из любой из них достижимы точки *A*, *B*, *C*, *D*. Второй центр составляет точка *E*, из которой достижимы точки *D*, *E* и *F*.

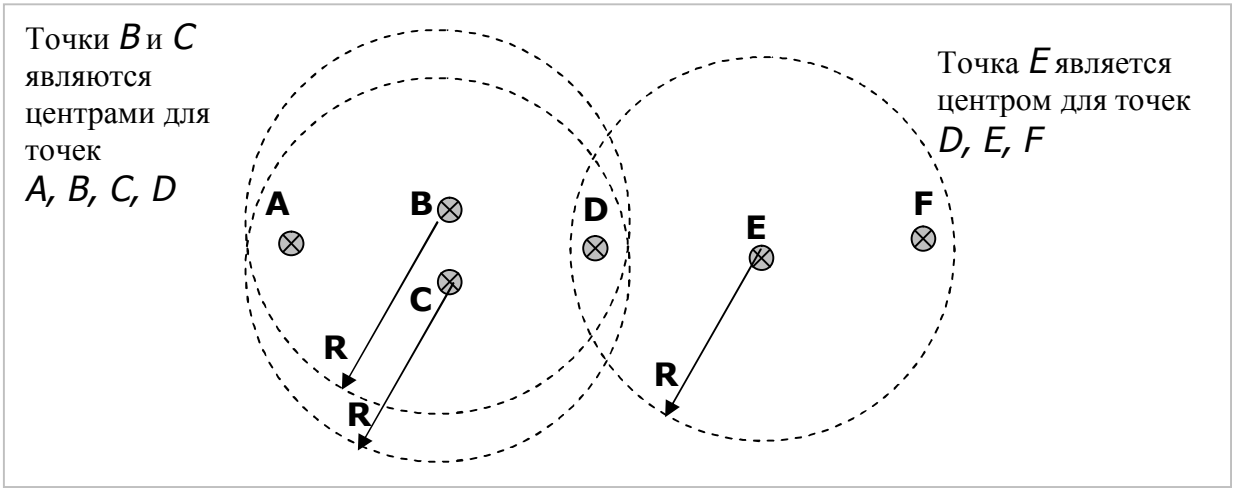


Рис. 21-3 — Левый центр (точки B и C) и правый центр (точка E)

Опираясь на эту геометрическую аналогию, опишем порядок поиска *внешних* p-центров графа с заданным взвешенным расстоянием **λ** (здесь **λ** является подобием радиуса **R**):

- Для каждой вершины графа построим *входящую* гамму, ограниченную взвешенным расстоянием **λ** (подобие круга).
- Последовательно сформируем набор из непустых пересечений гамм (множеств вершин), из которых достижимы: 1) отдельные вершины, 2) пары вершин, 3) тройки вершин, 4) четвёрки вершин и т.д. Если какое-то подмножество достигаемых вершин входит в более мощное, то подмножество отбрасываем.
- К полученному набору подмножеств достигаемых вершин применим алгоритм поиска наименьшего покрытия (ЗНП). Соответствующие этому покрытию множества *пересечений* и будут p-центрами графа для взвешенного расстояния **λ**.

Почти так же отыскиваются *внутренние* p-центры с тем отличием, что вначале строятся *исходящие* гаммы, ограниченные взвешенным расстоянием **λ.**

21.3. Решение прямой задачи

Перечисленные выше этапы решения прямой задачи рассмотрим на примере поиска p-центров в графе, показанном на рис. 21-4. Взвешенный радиус **λ** примем равным единице, веса вершин и длины рёбер этого графа также равны единице, что облегчает вычисления «в уме».

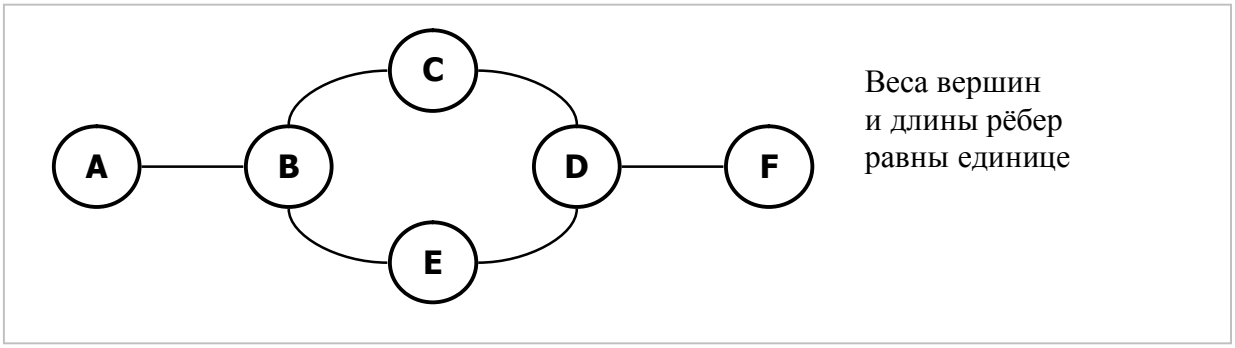


Рис. 21-4 — Пример графа с единичными длинами рёбер и единичными весами вершин

21.3.1. Формирование входящих гамм

Нам нужен объект (аналог круга) для хранения *ограниченной входящей гаммы* вершины вместе с самой вершиной, вот его объявление:

```
TGamma = class(TItem)
    mRoot  : TNode;  // достигаемая (центральная) вершина
    mGamma : TSet;   // вершины, из которых она достижима
    constructor Create(aRoot: TNode; aGamma: TSet);
    destructor Destroy;  override;
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;
```

Реализацию этого простого класса смотрите в листинге модуля **Graph**.

Найти каждую такую ограниченную гамму можно слегка модифицированным алгоритмом Дейкстры. Но для многократного поиска гамм (для разных вершин и для разных значений **λ**) гораздо выгодней предварительно построить карту дальних указателей, как мы сделали это в предыдущей главе при поиске центров. После формирования карты методом **InitMap**, все входящие гаммы вершин, ограниченные взвешенным расстоянием **λ**, строятся вспомогательными методами **GenLimitGamma** и **GenLimitGams** (первый метод вызывается из второго).

Листинг 21-1 — Функции формирования множества ограниченных входящих гамм

```
// Формирование множества вершин, из которых узел aNode
// достижим в пределах взвешенного расстояния aLambda
// (ограниченная входящая гамма)
// Применяется созданная ранее карта дальних указателей.

function TGraph.GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
var FL : TFarLink;     // очередная дальняя связь
begin
  Result:= CreateSet;
  with aNode.mFarLinks do begin  // просматриваем дальние связи вершины
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      if aLambda >= (FL.mDist * aNode.mValue) // если в достижимых пределах
        then Result.Insert(FL.mNodeFar);      // то вставляем в множество
```

```
      FL:= GetNext as TFarLink;
    end;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Построение множества ограниченных гамм типа TGamma

function TGraph.GenLimitGams(aLambda: integer): TSet;
var  Node : TNode;
     GammaSet: TSet;
     GammaItem: TGamma;
begin
  Result:= CreateSet;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    GammaSet:= GenLimitGamma(Node, aLambda);  // формируем огранич. вх. гамму
    GammaItem:= TGamma.Create(Node, GammaSet);// формируем пару вершина+гамма
    Result.Insert(GammaItem);
    Node:= NodeNext;
  end;
end;
```

Для рассматриваемого нами графа эти методы сформируют множество объектов класса **TGamma**, шесть таких объектов построены по данным рис. 21-4 и показаны в табл. 21-1 (это построение легко проверить в уме).

Табл. 21-1 — Области достижимости отдельных вершин при λ=1

Корневая вершина mRoot	Откуда достигается при λ=1, mGamma					
A	A	B				
B	A	B	C		E	
C		B	C	D		
D			C	D	E	F
E		B		D	E	
F				D		F

21.3.2. Формирование пересечений

На втором этапе создаём пересечения ограниченных гамм с одновременным формированием множеств достигаемых вершин. Для хранения этих объектов тоже потребуется особый класс, который построим на основе **TCostSet**, — объекта, используемого при поиске наименьших покрытий (ЗНП):

```
  TAreal = class(TCostSet)
    // поле mSet - достигаемые вершины, унаследовано от TCostSet
    mGamma : TSet;  // вершины, из которых достижимы вершины в mSet
    constructor Create(aAreal, aGamma: TSet);
    destructor Destroy;  override;
    procedure Print(var aFile: TextFile); override;
  end;
```

В поле **mSet** будем накапливать множества достигаемых вершин (одиноких, затем пар, затем троек, четвёрок и т.д.). Поле **mGamma** хранит пересечение, из которого достижимы эти вершины в **mSet**. Обратите внимание, что с разрастанием множества **mSet** (достигаемых вершин), сужается множество **mGamma**.

Для накопления объектов **TAreal** отлично подходит универсальный буфер **TBuffer**. Будем использовать его в режиме очереди (кольца), извлекая ранее созданные объекты функцией **Get**, и занося новые процедурой **Put**.

Далее я ссылаюсь на метод **TGraph.GenIntersections** (листинг 21-2). Он принимает предварительно построенное множество ограниченных входящих гамм. При инициализации в пустой пока буфер результата заносится фиктивный элемент, содержимое которого показано в табл. 21-2. Этот «затравочный» элемент служит основой для последующего формирования достигаемых вершин в первом внешнем цикле, и будет удалён в этом цикле.

Табл. 21-2 — Начальный фиктивный элемент TAreal

Достигаемые вершины mSet (пусто)						Откуда достигаются (λ=1) mGamma					
						A	B	C	D	E	F

Итак, *внешний* цикл извлекает и обрабатывает все имеющиеся на текущий момент элементы буфера. Напомню, что в первом цикле извлечём единственный фиктивный элемент. *Внутренний* цикл обработки элемента перебирает множества ограниченных гамм (из параметра **aGams**), пытаясь к текущему элементу поочерёдно «прилепить» ещё одну достигаемую вершину (к полю **mSet**). Присоединение возможно при двух условиях: 1) добавляемой вершины ещё нет в поле **mSet**, и 2) пересечение гаммы этой вершины с гаммой обрабатываемого элемента (полем **mGamma**) не пусто. В случае успеха создаётся и помещается в кольцевой буфер новый элемент **TAreal**. Поскольку по отношению к фиктивному элементу оба условия выполняются, первый внешний цикл пополнит кольцевой буфер восемью элементами, показанными в табл. 21-3. Это *одиночные* вершины, достигаемые из соответствующих им гамм.

Табл. 21-3 — Содержимое кольцевого буфера после 1-го цикла

Достигаемые одиночные вершины mSet						Откуда достигаются (λ=1) mGamma					
A						A	B				
	B					A	B	C		E	
		C					B	C	D		
			D					C	D	E	F
				E			B		D	E	
					F				D		F

А куда подевался фиктивный элемент? После извлечения из буфера он естественным образом исчез. Когда к элементу не «лепится» ни одной вершины, его надо вновь вернуть в буфер, но сейчас не тот случай.

Второй внешний цикл сформирует *пары* достигаемых вершин и соответствующие им гаммы (табл. 21-4). Принцип тот же, что и в первом цикле: добавляем к **mSet** отсутствующие вершины, когда пересечение гамм не пусто. Например, при добавлении к множеству **mSet**, состоящему из вершины *A* новой вершины *B* ищется пересечение их гамм:

$$(A\text{-}B) \bullet (A\text{-}B\text{-}C\text{-}E) = A\text{-}B$$

Заметим, что при добавлении к вершине *B* вершины *A* порождается такая же пара (дубликат) и засоряет буфер. Во избежание дубликатов перед созданием нового элемента проверяется, создавался ли он ранее? Для проверки введено вспомогательное множество **Existed**, хранящее ранее созданные достигаемые подмножества.

Табл. 21-4 — Содержимое кольцевого буфера после 2-го цикла

Достигаемые пары вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B					A	B				
A		C					B				
A				E			B				
	B	C					B	C			
	B		D					C			
	B			E			B			E	
		C	D					C	D		
		C		E			B		D		
		C			F				D		
			D	E					D	E	
			D		F				D		F
				E	F				D		

Обратите внимание, что в буфере остались только *пары* достигаемых вершин, а *одиночные* при извлечении автоматически покинули буфер, они не возвращены в него, поскольку являются подмножествами пар.

В третьем внешнем цикле обрабатываем пары и формируем элементы с тройками достигаемых вершин (табл. 21-5).

Табл. 21-5 — Содержимое кольцевого буфера после 3-го цикла

Достигаемые тройки вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B	C					B				
A	B			E			B				
	B	C	D					C			
	B	C		E			B				
	B		D	E						E	
		C	D		F				D		
		C	D	E					D		
			D	E	F				D		

Наконец в четвёртом цикле найдутся области из четырёх достигаемых вершин, их окажется лишь две. К ним добавятся и две тройки (*B-C-D* и *B-D-E*), поскольку тройки не удалось расширить, и они не вошли подмножествами в другие множества. По этой причине тройки возвращены в кольцевой буфер.

Табл. 21-6 — Содержимое кольцевого буфера после 4-го цикла

Достигаемые тройки и четвёрки вершин mSet						Откуда достигаются (λ=1) mGamma					
A	B	C		E			B				
	B	C	D					C			
	B		D	E						E	
		C	D	E	F				D		

В пятом цикле ни одной вершины не присоединится, и потому кольцевой буфер не изменится, — на этом процедура завершится.

Готовый буфер «скармливаем» затем функции **`CollectMinCover`** для поиска наименьшего покрытия. В табл. 21-6 решение выделено цветом, здесь для **λ**=1 найдено два центра в вершинах *B* и *D* (рис. 21-5).

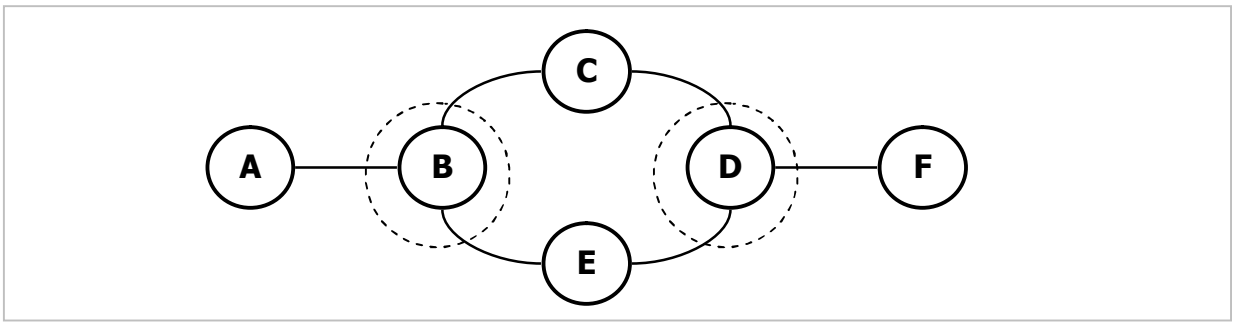


Рис. 21-5 — Два центра графа для λ =1

Построив такие же таблицы для взвешенного расстояния **λ**=2, получим один центр, состоящий из двух вершин, как показано на рис. 21-6. Здесь расстояние к любой вершине графа от вершин *C* и *E* составляет не более 2-х единиц.

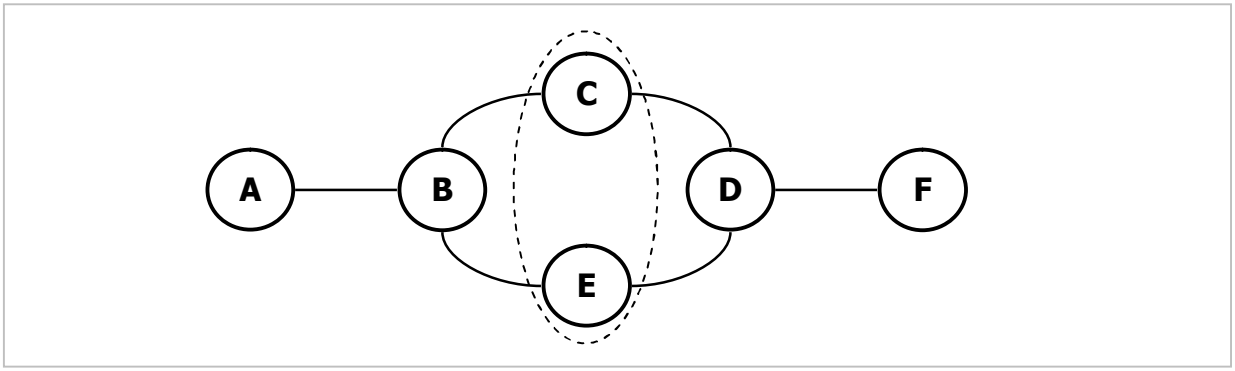


Рис. 21-6 — Один центр (из двух вершин) для λ =2

Листинг 21-3 содержит метод **GenPCentLambda**, решающий прямую задачу поиска p-центров по заданному значению **λ:**

Листинг 21-2 — Формирование всех возможных пересечений гамм

```
function TGraph.GenIntersections(aGams: TSet): TBuffer;
var
    GammaItem : TGamma;    // элемент для входящих гамм
        Nodes : TSet;      // достигаемые вершины (TAreal.mSet)
     GammaSet : TSet;      // откуда достигаются (TAreal.mGamma)
         Temp : TSet;      // здесь вычисляется пересечение гамм
      Existed : TSet;      // множество множеств уже обработанных областей
        Areal : TAreal;    // очередной элемент "область"
         Flag : boolean;   // признак пополнения буфера
          Cnt : integer;   // текущее количество элементов в буф. результ.
            i : integer;   // индекс
begin
  Result:= TBuffer.Create;  // буфер результата
  Temp:= CreateSet;           // здесь вычисляется пересечение гамм
  Existed:= CreateSet;        // множество множеств уже обработанных областей
  Nodes:= CreateSet;          // достигаемые вершины (TAreal.mSet)
  GammaSet:= CreateSet;       // откуда достигаются (TAreal.mGamma)
  GammaSet.CopyItems(mNodes); // копия множества всех вершин
  // Создаём и помещаем в буфер фиктивный элемент TAreal такой, что:
  // mNodes - множество достигаемых вершин ПУСТО
  // mGamma - множество вершин, из которых достигается, содержит ВСЕ вершины
  Result.Put(TAreal.Create(Nodes, GammaSet));
  repeat
    Cnt:= Result.GetCount;  // текущее количество элементов в буфере
    for i:=1 to Cnt do begin
      // Цикл по всем элементам буфера
      Flag:= false;                // сброс признака пополнения буфера
      Areal:= Result.Get as TAreal; // очередной элемент из буфера
      // Перебор всех элементов исходного множества
      GammaItem:= aGams.GetFirst as TGamma;
      while Assigned(GammaItem) do begin
        if not Areal.mSet.Exist(GammaItem.mRoot) then begin
          // Если текущая область ещё не содержит корневой вершины
          // то вычисляем пересечение входящих гамм
          Temp.CopyItems(Areal.mGamma);     // гамма в текущей области
```

```
          Temp.Mul(GammaItem.mGamma);       // пересекаем с гаммой вершины
          if Temp.GetCount <> 0 then begin
            // если пересечение не пусто,
            // пытаемя создать и добавить в буфер новую область
            Nodes:= CreateSet;                // достигаемые (TAreal.mSet)
            Nodes.CopyItems(Areal.mSet);      // существующие вершины
            Nodes.Insert(GammaItem.mRoot);    // плюс ещё одна вершина
            if Existed.Exist(Nodes) then begin
              // Если это подмножество достигаемых уже существует в наборе
              Nodes.Free;     // то удаляем дубликат
              Nodes:= nil;    // и ничего не вставляем
            end else begin
              // Если получено новое подмножество достигаемых вершин
              Existed.Insert(Nodes);      // запоминаем для будущих проверок
              // создаём новую область и вставляем в буфер
              GammaSet:= CreateSet;      // откуда достигаются (TAreal.mGamma)
              GammaSet.CopyItems(Temp); // копия пересечения GammaSet= Temp
              Result.Put(TAreal.Create(Nodes, GammaSet));
            end; // if
            Flag:= true;  // установить признак пополнения буфера
          end; // if
        end; // if
        GammaItem:= aGams.GetNext as TGamma;  // перебор множества гамм
      end; // while
      // После обработки очередного элемента буфера проверяем:
      // Если обработанный элемент буфера породил хотя бы один новый элемент,
      // то он является подмножеством этого нового и исключается
      // из дальнейшей обработки, а иначе возвращается в буфер
      if Flag then begin
        Existed.Delete(Areal.mSet);// удаляем достигаемые из проверяемых
        Areal.Free;                // удаляем ненужный элемент буфера
      end else begin
        Result.Put(Areal);         // а иначе возвращаем элемент в буфер
      end;
    end; // for
    // выход, если буфер не изменился
  until not Flag and (Cnt=Result.GetCount);
  Temp.Free;
  Existed.Free;
end;
```

Листинг 21-3 — Метод, решающий прямую задачу нахождения P-центров

```
// Поиск множества центров
// с заданной константой проникновения aLambda (дистанцией)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
var   Gams : TSet;
       Buf : TBuffer;
begin
  // Формирование карты дальних указателей (направление карты инверсно):
  case aDirect of
    InCenter:    InitMap(OutCenter);
    OutCenter:   InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование входящих гамм
  // с заданной константой проникновения (дистанцией) aLambda
```

```
  Gams:= GenLimitGams(aLambda);
  // Формирование всех непустых пересечений
  Buf:= GenIntersections(Gams);
  // Решение задачи о минимальном покрытии (ЗНП)
  Result:= CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  RemoveItemsFromBuf(Buf, Result.mSet);
  // Освобождаем множество входящих гамм и буфер
  Gams.ClrAndDestroy;
  Gams.Free;
  Buf.ClrAndDestroy;
  Buf.Free;
end;
```

Обратите внимание, что перед вызовом метода **InitMap** параметр инвертируется, поскольку для поиска *внешних* p-центров строится карта, используемая при поиске *внутреннего* 1-центра, и наоборот.

21.4. Решение обратной задачи

Напомню формулировку обратной задачи о p–центрах: дана кратность центров **p**, и надо найти эти центры — вершины графа, а также соответствующий им взвешенный радиус **λ**.

Цель достигается многократным решением прямой задачи с растущим значением **λ**. Так, при **λ=0** количество центров будет равно количеству вершин графа. По мере роста **λ** численность центров снижается, вплоть до единицы. Когда количество центров достигнет нужного значения **p**, задача решена. Вопрос лишь в том, с каким шагом наращивать **λ**? Делая это слишком осторожно, мы обречём себя на лишнюю работу, поскольку далеко не каждое приращение радиуса сокращает численность центров. Предположим, мы нашли все *взвешенные* расстояния между всеми вершинами графа и расположили эти числа без повторений в порядке возрастания. Тогда очередное значение **λ** можно выбирать из этого ряда, поскольку промежуточные значения **λ** не уменьшат число центров. Итак, **λ** выбираем из множества **Lambdes**, заранее сформированного локальной процедурой **LambdasCreate**.

Листинг 21-4 — Решение обратной задачи поиска p-центров

```
function TGraph.GenPCenters(aPoly: integer;       // кратность центров
                            var aLambda: integer;  // радиус
                            aCenter: TCenter // InCenter,OutCenter,InOutCenter
                            ): TCostSet;

var Lambdes : TSet;  // лямбда-список (возрастающее множество расстояний)
       LRec : TSortedNum; // элемент лямбда-списка
     Lambda : integer;    // текущий взвешенный радиус
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества взвешенных расстояний (уникальных)
  procedure LambdasCreate;

    procedure AddLambdas(aNode: TNode);
    var FL : TFarLink;    // очередная дальняя связь
```

```
        SN : TSortedNum;  // элемент сортированного списка чисел
        Lambda: integer;  // взвешенное расстояние от текущей вершины
    begin
      with aNode.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          Lambda:= FL.mDist * aNode.mValue;
          SN:= TSortedNum.Create(Lambda);
          if not Lambdes.Insert(SN) then SN.Free;
          FL:= GetNext as TFarLink;
        end;
      end;
    end;
  var  Node : TNode;    // текущая вершина
  begin { LambdasCreate }
    Lambdes:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // перебор вершин
      AddLambdas(Node);             // обработать вершину
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPCenters }
  // Формирование карты дальних указателей:
  case aCenter of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование лямбда-списка (возрастающего множества расстояний)
  LambdasCreate;
  // Выбор первого (наименьшего) элемента взвешенного расстояния
  LRec:= Lambdes.GetFirst as TSortedNum;
  repeat
    Lambda:= LRec.mNumber;            // очередное взвешенное расстояние
    Result:= GenPCentLambda(Lambda, aCenter);  // множество центров
    Writeln('Lambda= ', Lambda:3, '  P= ', Result.mSet.GetCount:3);
    // Если количество центров достигнуто, то выход:
    if Result.mSet.GetCount <= aPoly then Break;
    // а иначе очищаем результат и берём следующий по величине радиус
    Result.ClrAndDestroy;
    Result.Free;
    Result:= nil;
    // Выбор очередного элемента взвешенного радиуса (по возрастанию)
    LRec:= Lambdes.GetNext as TSortedNum;
  until not Assigned(LRec);
  // Возвращаем взвешенный радиус:
  aLambda:= Lambda;
  // Очистка и уничтожение списка расстояний
  Lambdes.ClrAndDestroy;
  Lambdes.Free;
  // Очистка карты дальних указателей:
  DoneMap;
end;
```

21.5. Испытания

21.5.1. Прямая задача

Ниже дана программа для тестирования решения прямой задачи.

Листинг 21-5 — Программа поиска P-центров по заданному взвешенному расстоянию λ

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas',
  Assembly in '..\Common\Assembly.pas';

var Gr: TGraphChars;
    PCenters: TCostSet;
    Lambda : integer;
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  repeat
    Write('Lambda= '); Readln(Lambda);
    if Lambda = 0 then Break;
    PCenters:= Gr.GenPCentLambda(Lambda, OutCenter);
    if Assigned(PCenters) then begin
      Writeln('Lambda = ', Lambda);
      PCenters.Expo;
      PCenters.ClrAndDestroy;
      PCenters.Free;
    end;
  until false;
  Gr.Free; // Освобождение графа
end.
```

Для испытания воспользуемся графом, показанным на рис. 21-7.

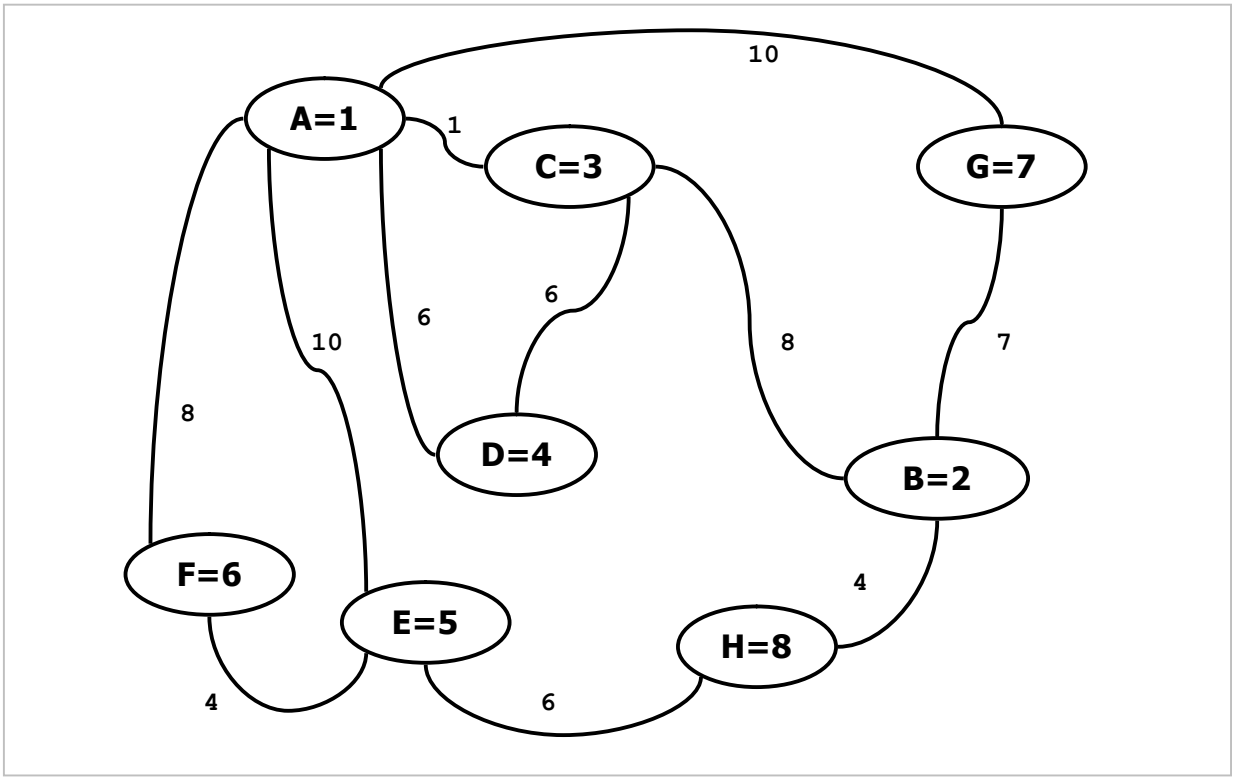


Рис. 21-7 — Взвешенный неориентированный граф

Текст для ввода графа таков:

```
Test.txt - Взвешенный неориентированный граф
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Протокол работы программы представлен ниже. Для **λ=50** найдены два центра (A и B). Слева от стрелки ==> указаны центральные вершины, а справа — вершины, достигаемые из этих центров с заданным взвешенным расстоянием.

```
Lambda = 50
2 =
{
A ==> A,B,C,D,E,F
B ==> A,B,C,E,G,H
} : 2
```

21.5.2. Обратная задача

Здесь представлена программа для тестирования обратной задачи — поиска p-центров и взвешенного радиуса по заданному количеству центров **P**.

Листинг 21-6 — Программа поиска заданного количества p-центров и взвешенного расстояния λ

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr: TGraphChars;
    PCenters: TCostSet;
    P : integer;
    Lambda : integer;
    Start, Time : TDateTime;
    D : char;
    Direct : TCenter;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    case Direct of
      InCenter: Writeln('------------------- IN --------------------');
      OutCenter: Writeln('------------------- OUT -------------------');
      InOutCenter: Writeln('------------------- IN-OUT ----------------------');
    end;
    Start:= Now;
    PCenters:= Gr.GenPCenters(P, Lambda, Direct);
    Time:= MilliSecondsBetween(Start, Now);
    if Assigned(PCenters) then begin
      Writeln('- - - - - - - - - - - - - - - - - - -');
      PCenters.Expo;
      Writeln('Lambda= ', Lambda);
      Writeln('Time (ms) = ', Time:6:0);
      PCenters.ClrAndDestroy;
      PCenters.Free;
    end;
  until false;
end.
```

Протокол работы программы дан ниже. Промежуточные значения **Lambda** и **P** печатаются здесь для наглядности внутри метода **GenPCenters**, после испытаний эту печать надо отключить.

```
Test_4 – (P-центры)
{
A =  1 ->   { C= 1 D= 6 E= 10 F= 8 G= 10 } : 5
B =  2 ->   { C= 8 G= 7 H= 4 } : 3
C =  3 ->   { A= 1 B= 8 D= 6 } : 3
D =  4 ->   { A= 6 C= 6 } : 2
E =  5 ->   { A= 10 F= 4 H= 6 } : 3
F =  6 ->   { A= 8 E= 4 } : 2
G =  7 ->   { A= 10 B= 7 } : 2
H =  8 ->   { B= 4 E= 6 } : 2
} : 8
- - - - - - - - - - - - - - - - - - -
Кратность центров P= 2
Lambda=  0  P=  8
Lambda=  1  P=  7
Lambda=  3  P=  7
Lambda=  6  P=  7
Lambda=  8  P=  6
Lambda=  9  P=  6
Lambda= 10  P=  6
Lambda= 13  P=  6
Lambda= 14  P=  6
Lambda= 16  P=  6
Lambda= 18  P=  5
Lambda= 20  P=  4
Lambda= 24  P=  4
Lambda= 27  P=  4
Lambda= 28  P=  4
Lambda= 30  P=  4
Lambda= 32  P=  4
Lambda= 33  P=  4
Lambda= 36  P=  4
Lambda= 48  P=  3
Lambda= 49  P=  3
Lambda= 50  P=  2
- - - - - - - - - - - - - - - - - - -
{
A ==> A,B,C,D,E,F
B ==> A,B,C,E,G,H
} : 2
Lambda= 50
```

Для 2-центра графа, показанного на рис. 21-7, найдены вершины *A* и *B*, при этом взвешенный радиус составил 50 единиц. В табл. 21-7 даны результаты поиска ещё нескольких p-центров этого графа.

Табл. 21-7 — P-центры графа, показанного на рис. 21-7

Кратность центров P	Вершины центров	Взвешенный радиус λ
1	H	77
2	A, B	50
3	A, E, G	48
4	D, F, G, H	20
5	D, E, F, G, H	18

Следующее испытание проведено на не взвешенном орграфе, который показан на рис. 21-8.

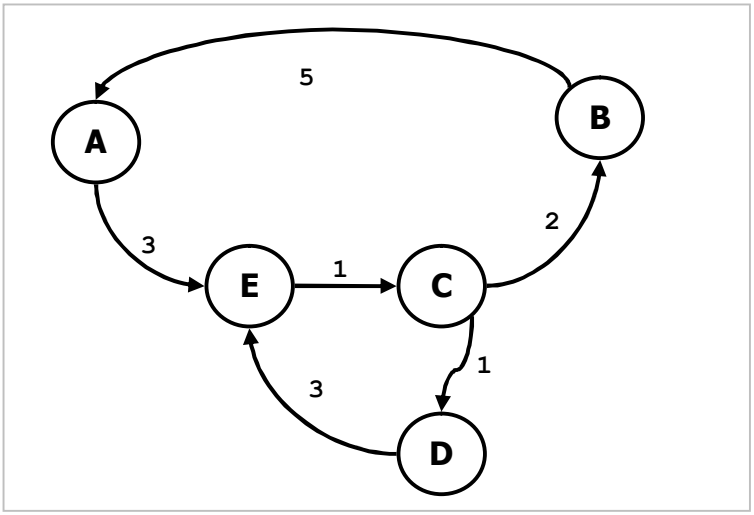


Рис. 21-8 — Не взвешенный орграф

Результаты представлены в табл. 21-8.

Табл. 21-8 — Результаты поиска p-центров на не взвешенном орграфе

Кратность центров P	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Радиус λ	Вершины	Радиус λ	Вершины	Радиус λ
1	A	6	B	6	C E	11
2	A, E	3	B, E	3	-	-
3	A, C, E	2	A, B, D	2	A-B-C, A-B-D, A-B-E	5
4	A, B, C, E	1	A, B, C, D	1	-	-

Здесь не существуют внешне-внутренние 2-центр и 4-центр. Это значит, что расширение множества достигаемых вершин с одной до двух, и с трёх до четырёх не уменьшает взвешенного радиуса графа. Наконец, ниже даны исходные данные и результаты поиска центров на взвешенном орграфе, показанном на рис. 21-9.

```
Взвешенный орграф (рис. 21-9)
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

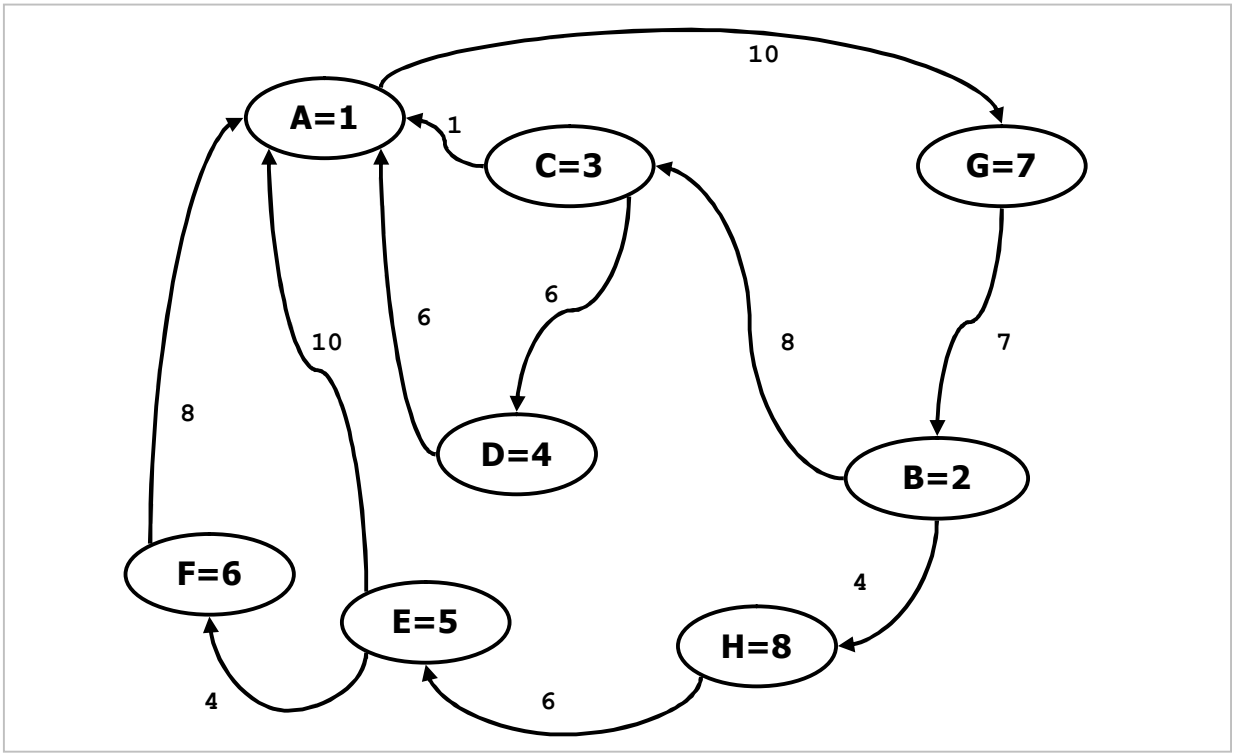


Рис. 21-9 — Взвешенный орграф

Табл. 21-9 — Р-Центры взвешенного орграфа (рис. 21-9)

Кратность Р	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	λ	Вершины	λ	Вершины	λ
1	G	93	A	128	A, B, E, G, H	296
2	C, H	77	A, H	77	A-H, B-H, G-H	234
3	B, E, G	56	A, E, G	48	A-F-H, B-F-H, G-F-H	185
4	C, E, G, H	24	D, F, G, H	20	E, F, G, H	148
5	-	-	D, E, F, G, H	18	D, E, F, G, H	78

Здесь отметим отсутствие внешнего 5-центра, а также наличие нескольких вариантов внешне-внутренних 1-, 2- и 3-центров.

21.6. Итоги

21.6.1. Задача о p-центрах заключается в поиске заданного количества центров, то есть, подмножеств таких вершин, расстояние от которых до самых дальних вершин минимально. Каждый из P центров в общем случае является подмножеством вершин. Различают прямую и обратную задачу о p-центрах.

21.6.2. В прямой задаче задан взвешенный радиус λ и требуется найти множество удовлетворяющих ему центров.

21.6.3. В обратной задаче известна кратность центров P, и надо найти их распределение по вершинам и соответствующий минимальный взвешенный радиус λ.

21.6.4. Обратная задача о p-центрах решается через многократное решение прямой задачи с возрастающим значением λ.

21.7. Задания

21.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 22
P-медианы

22.1. Медианы и p-медианы

При размещении *экстренных* служб стремятся уменьшить *максимальные* расстояния от центра к периферии. Теперь позаботимся об экономии, и постараемся снизить *сумму* взвешенных расстояний ко всем вершинам, — такие задачи называют *минисуммными*. Вот несколько примеров этого рода.

Пусть островной архипелаг и мосты образуют неориентированный связанный граф. Поищем остров для постройки электростанции с тем, чтобы минимизировать общую длину линий электропередач, протянутых от неё к другим островам по существующим уже мостам. Искомая вершина графа называется *медианой*. Обычно острова разнятся уровнем потребления энергии, а это влияет на толщину проводов, и тогда затраты станут пропорциональны как расстоянию до острова, так и его «аппетиту» (весу вершины), — здесь надо свести к минимуму общую массу проводов.

Теперь обратимся к орграфу, где архипелаг соединён мостами с односторонним движением. Пусть эта связь будет сильной, и гарантирует взаимную достижимость любой пары островов. Поищем место для базы снабжения товарами так, чтобы минимизировать затраты на их доставку от базы к потребителям. Здесь не учитываем затраты на возврат порожнего транспорта (обратный маршрут порожняка в общем случае отличается от прямого). Если населённость островов различна, то количество доставляемых товаров зависит от веса вершины-адресата, и тогда следует минимизировать *сумму взвешенных* расстояний от искомой вершины к остальным, — искомая вершина называется тут *внешней медианой* орграфа.

Теперь найдём место заводу для переработки отходов. Здесь груз доставляется в обратном направлении — от периферии к центру переработки, или точнее к вершине, называемой *внутренней медианой* орграфа.

Наконец, при выборе столичного острова желательно минимизировать транспортные издержки для населения (в столицу и обратно ездят все). Здесь минимальные транспортные затраты соответствуют вершине, называемой *внешне-внутренней медианой* орграфа.

Пока шла речь о единственной искомой вершине, но для нескольких пунктов обслуживания надо говорить (как и для центров), о кратных *p-медианах*, где *p* — кратность медианы. Разумеется, что если пункты разбросать по графу равномерно, это снизит сумму взвешенных расстояний. Тогда любую вершину, не входящую в подмножество медианы, надо *прикрепить* к *ближайшей* из этого подмножества. Для орграфов *кратные* p-медианы могут быть *внешними*, *внутренними* и *внешне-внутренними*.

Все эти задачи о медианах и кратных p-медианах решаются, к счастью, одним алгоритмом, который мы далее и построим. Существуют задачи, где надо учесть ещё и затраты на постройку или эксплуатацию самой службы (когда эти затраты различны для различных вершин). Их решение достигается вводом дополнительного поля в вершину графа и небольшой модификацией основного алгоритма, — постройте такой метод сами.

22.2. Немного формализма

Возьмём *i*-ю вершину орграфа и определим для неё сумму взвешенных расстояний ко всем остальным вершинам орграфа, эта характеристика вершины называется *внешним передаточным числом*:

$$\sigma_o(x_i) = \sum_{x_j \in X} v_j \bullet d(x_i, x_j)$$

Схожим образом определяется *внутреннее передаточное число*, но здесь суммируются расстояния от всех вершин к *i*-й вершине:

$$\sigma_t(x_i) = \sum_{x_j \in X} v_j \bullet d(x_j, x_i)$$

Здесь X — это множество всех вершин графа, v_j — вес j–й вершины, $d(x_i, x_j)$ — расстояние от i–й вершины к j–й, а $d(x_j, x_i)$ — расстояние от j–й вершины к i–й (обратное).

Вершина, претендующая на роль внешней или внутренней медианы, должна обладать подобающими минимальными передаточными числами. Для внешне-внутренней медианы минимальной должна быть сумма двух её передаточных чисел. Стало быть, задача поиска единственной медианы решается перебором всех вершин графа, с сопутствующим вычислением передаточных чисел.

Сложнее обстоит дело с кратными p-медианами, где передаточные числа определяются так:

$$\sigma_o(X_p) = \sum_{x_j \in X} v_j \bullet d(X_p, x_j)$$

$$\sigma_t(X_p) = \sum_{x_j \in X} v_j \bullet d(x_j, X_p)$$

Здесь Xp — некое подмножество из p вершин, $d(Xp,x_j)$ — расстояние от ближайшей вершины этого подмножества к j–й вершине, $d(x_j,Xp)$ — расстояние от j–й вершины к ближайшей из Xp подмножества.

При поиске кратных p-медиан надо перебрать все комбинации из p вершин. Для 1-медианы алгоритм вырождается в линейный перебор вершин, но для значений p, близких к $N/2$, задача обретает экспоненциальную сложность, что вынуждает искать пути сокращения перебора.

22.3. Числовые примеры

Найдём положение медиан в графе, показанном на рис. 22-1.

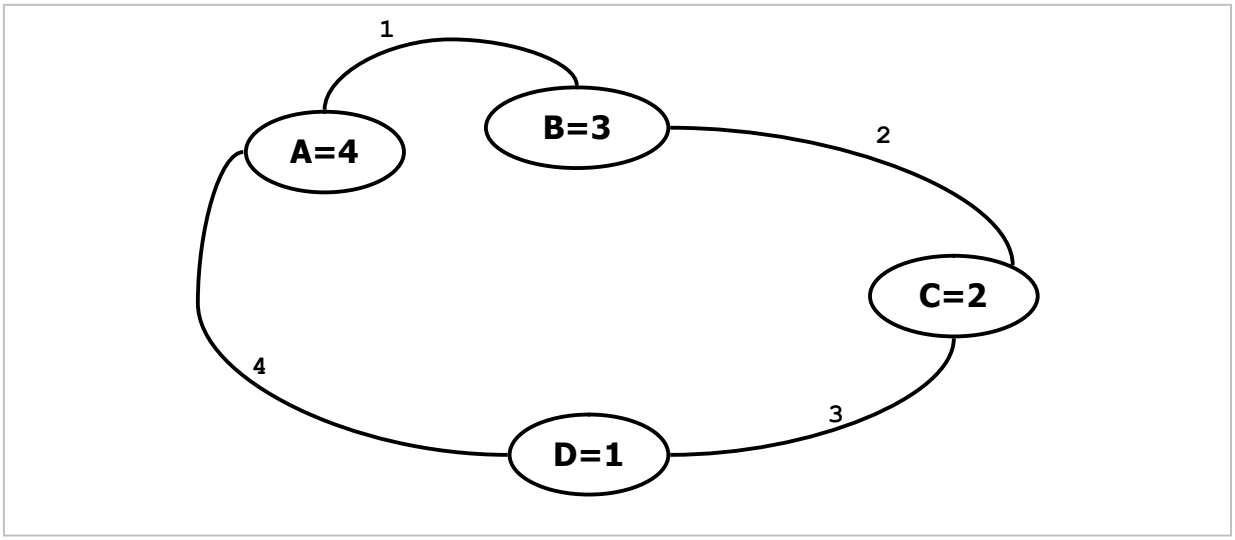


Рис. 22-1 — Неориентированный граф с взвешенными дугами и вершинами

По рисунку составим вспомогательную таблицу (карту) взаимных расстояний $d(i,j)$ между вершинами x_i и x_j (табл. 22-1).

Табл. 22-1 — Взаимные расстояния d(i,j) между i-й и j-й вершинами

Позиция	Исходные вершина Xi							
	A		B		C		D	
	Xj	d	Xj	d	Xj	d	Xj	d
1	A	0	B	0	C	0	D	0
2	B	1	A	1	B	2	C	3
3	C	3	C	2	A	3	A	4
4	D	4	D	5	D	3	B	5

В левом столбце указан порядковый номер целевой j–й вершины относительно исходной i–й вершины. Первой в списке идёт исходная вершина, поскольку расстояние к ней равно нулю. Прочие вершины следуют далее в неубывающем порядке расстояний $d(i,j)$.

По таблице легко подсчитать сумму взвешенных расстояний (передаточные числа) для любой вершины; так, к примеру, для вершины A эта сумма составит:

```
S = 4•0 + 3•1 + 2•3 + 1•4 = 13
```

Здесь первый множитель — вес *j*–й вершины, а второй — расстояние от *i*–й к *j*–й вершине. Медианой будет та из них, для которой сумма окажется минимальной (таких вершин может быть несколько), в данном случае это вершина *A*.

Для поиска 2-медиан надо перебрать все комбинации из *двух* вершин и вычислить аналогичные суммы с учётом того, что каждая вершина, не принадлежащая медиане, *прикрепится* к ближайшей из *двух* медианных вершин. Сами медианные вершины не вносят вклад в сумму, поэтому слагаемые с их участием отброшены. Например, если центры обслуживания разместить в вершинах *A* и *B*, то сумма взвешенных расстояний подсчитывается только для вершин *C* и *D* и составит:

```
S = 2•2 + 1•4 = 8
```

Сходно ищутся все последующие кратные медианы, в табл. 22-2 представлены 1-, 2- и 3-медианы рассмотренного графа.

Табл. 22-2 — Кратные медианы графа на рис. 22-1

Кратность P	Вершины медианы	Стоимость
1	A	13
2	A, C	6
3	A, B, C	3

Теперь рассмотрим поиск медиан в орграфе, представленном на рис. 22-2.

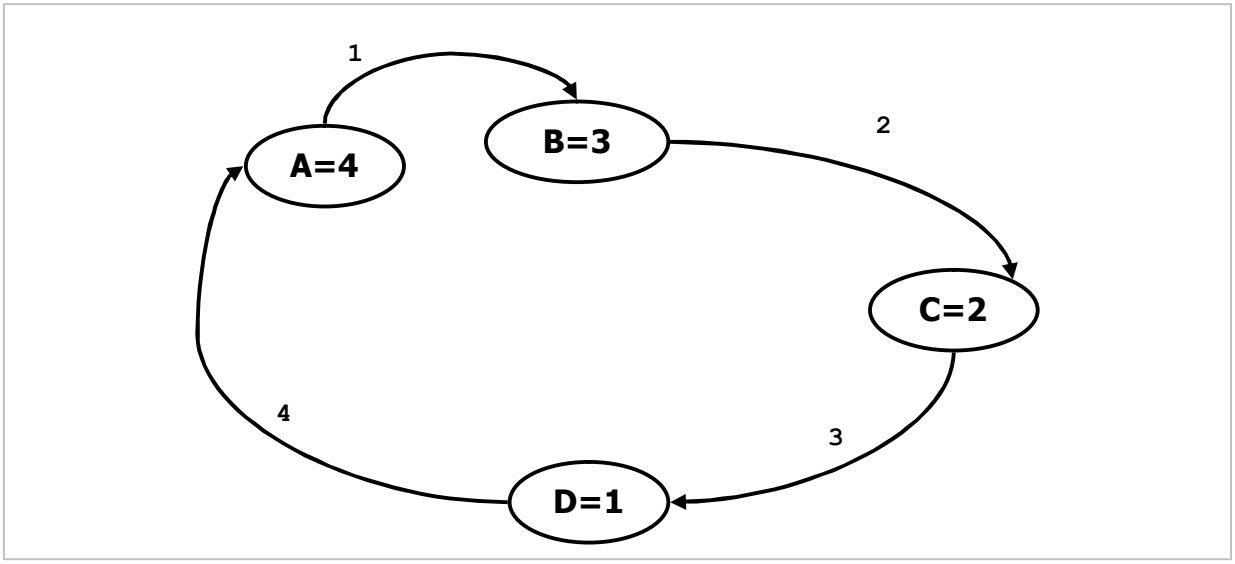


Рис. 22-2 — Орграф с взвешенными дугами и вершинами

Напомню, что в ориентированном графе существуют три рода p-медиан: внешние, внутренние и внешне-внутренние. Для поиска каждой из них надо

построить свою таблицу взаимных расстояний. Таблица для поиска внешних медиан строится по направлению дуг (табл. 22-3).

Табл. 22-3 — Взаимные расстояния d(i,j) между i-й и j-й вершинами

Позиция	Исходные вершина Xi							
	A		B		C		D	
	Xj	d	Xj	d	Xj	d	Xj	d
1	A	0	B	0	C	0	D	0
2	B	1	C	2	D	3	A	4
3	C	3	D	5	A	7	B	5
4	D	6	A	9	B	8	C	7

Для поиска внутренних медиан таблица строится в направлении, обратном направлению дуг (табл. 22-4).

Табл. 22-4 — Взаимные расстояния d(j,i) между j-й и i-й вершинами

Позиция	Исходные вершина Xj							
	A		B		C		D	
	Xi	d	Xi	d	Xi	d	Xi	d
1	A	0	B	0	C	0	D	0
2	D	4	A	1	B	2	C	3
3	C	7	D	5	A	3	B	5
4	B	9	C	8	D	7	A	6

Далее ход решения ничем не отличается от описанного выше: перебираются все возможные комбинации из p вершин. Результат поиска представлен в табл. 22-5.

Табл. 22-5 — P-медианы графа на рис. 22-2

Кратность P	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Стоимость	Вершины	Стоимость	Вершины	Стоимость
1	A	15	B	25	A	60
2	A-C	6	B-C	9	A-B	30
3	A-B-C	3	A-B-C	4	A-B-C	10

22.4. Основные идеи

Итак, упрощённый способ поиска p-медиан состоит в решении двух подзадач:

- перебор всех комбинаций из N вершин по P, где N — количество вершин графа;
- подсчёт для каждой комбинации её стоимости и сохранение комбинации с наименьшей стоимостью.

Поскольку для значений P, близких к $N/2$, число комбинаций растёт с ростом N экспоненциально, в улучшенном алгоритме будет решена подзадача, отсекающая заведомо тупиковые ветви перебора. Отсюда следует план дальнейших действий:

- создать алгоритм перебора всех комбинаций из N вершин по P;
- создать функцию вычисления стоимости текущей комбинации медианных вершин (текущего p-подмножества);
- на основе двух предыдущих этапов, разработать упрощённый метод поиска p-медиан *полным перебором* всех p-подмножеств;
- разработать алгоритм *отсечения* заведомо тупиковых ветвей, и тем ускорить перебор.

22.5. Принцип перебора комбинаций

Идея перебора без повторений всех комбинаций из N вершин по P показана на рис. 22-3. Мы видим «горнолыжную трассу», где воображаемый лыжник стартует с одной из вершин верхнего (нулевого) уровня, и спускается в коридоре, ширина которого определена выражением:

$$NN = N + 1 - P$$

Здесь N — число вершин графа, а P — мощность формируемого подмножества, которая, заметим, соответствует глубине спуска. Из формулы следует, что чем глубже спуск, тем уже коридор для манёвра.

На спуске лыжник маневрирует в направлении стрелок: при переходе на следующий уровень он может выбрать только один из узлов, что находится правее от него, но в пределах заданного коридора. По ходу спуска посещённые вершины накапливаются в очередном p-подмножестве. Накопление завершается на уровне $Level = p-1$ (здесь нумерация уровней $Level$ идёт с нуля), — на этом уровне подмножество из P элементов сформировано.

Отметим два крайних случая. При $P=1$ ширина коридора максимальна и составляет N, а глубина спуска минимальна, — тут алгоритм вырождается в линейный перебор всех вершин на верхнем уровне. Алгоритм вырождается в линейный и при $P=N$: здесь коридор сужается до единицы ($NN=1$), но зато лыжник достигает самого «подножья горы».

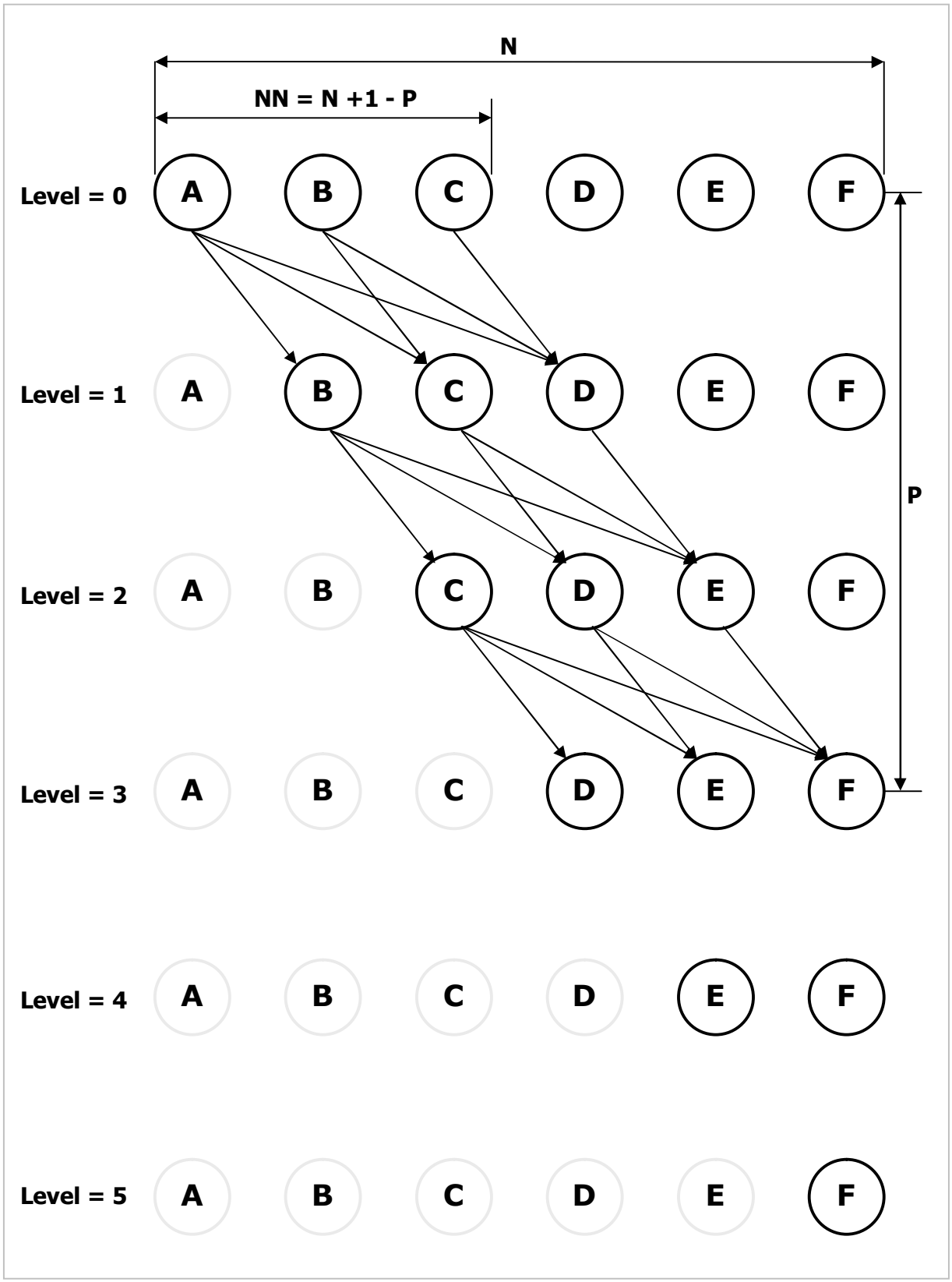


Рис. 22-3 — Порядок обработки вершин при формировании всех подмножеств из шести вершин по четыре (N=6, P=4)

Алгоритм реализован рекурсивной процедурой, для демонстрации которой служит следующая программа.

Листинг 22-1 — Программа для демонстрации перебора комбинаций

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

// Наследник графа с дополнительным демонстрационным методом

type  TGrTest = class(TGraphChars)
        procedure GenCombin(aPole: integer);
      end;

////////////////////////////////////////////////////////////
// Метод перебора всех комбинаций по P вершин

procedure TGrTest.GenCombin(aPole: integer);
var
    S: TSet;           // накопитель комбинации
    Level : integer;   // текущий уровень - 1
    NN: integer;       // количество обрабатываемых узлов на уровне
    Count : integer;   // счётчик комбинаций

  // Локальная рекурсивная процедура перебора вершин
  // aIndex - начальный индекс для перебора
  procedure Local(aIndex: integer);
  var i: integer;
      N: TNode;         // текущая вершина
  begin
    for i:= aIndex to NN do begin
      // очередная вершина на уровне:
      N:= mNodes.GetItem(i + Level) as TNode;
      S.Insert(N);         // вставляем в комбинацию
      if Level+1=aPole     // комбинация готова?
        then begin
          S.Expo;          // да, показать
          Inc(Count);      // и нарастить счётчик комбинаций
        end
        else begin
          Inc(Level);      // нет, нарастить уровень
          Local(i);        // и вызвать эту процедуру рекурсивно
          Dec(Level);      // возврат на текущий уровень
        end;
      S.Delete(N);         // удалить вершину из комбинации
    end;
  end;

begin
  S:= CreateSet;    // множество для формирования комбинаций
  Count:=0;         // счётчик комбинаций
  Level:= 0;        // текущий уровень
  // Количество узлов, обрабатываемых на каждом уровне:
  NN:= mNodes.GetCount + 1 - aPole;
  Local(1);         // вызов локальной процедуры перебора
  Writeln('Count= ', Count);
  S.Free;
end;
```

```
var Gr: TGrTest;
    P : integer;

begin
    Gr:= TGrTest.GenRandom(false, 10, 10, 7, 12);
    Gr.Expo;
    Writeln('- - - - - - - - - - - - - - - - - - -');
    repeat
      Write('P= '); Readln(P);
      if P=0 then Break;
      Gr.GenCombin(P);
      Writeln('- - - - - - - - - - - - - - - - - - -');
    until false;
end.
```

Чтобы не затрагивать основной объект (**TGraph**), здесь создан наследник графа с методом **GenCombin**, перебирающий все подмножества из *P* вершин. В счётчике **Count** подсчитывается общее количество комбинаций. Представленный демонстрационный метод послужит основой для «боевых» методов поиска медиан.

22.6. Вычисление стоимости

Перед рассмотрением функции, вычисляющей стоимость p-подмножества, условимся о передаче данных. Мы не станем передавать ей подмножество в явном виде, а просто окрасим *p* вершин графа чёрным цветом, отмечая тем самым места размещения служб. Разумеется, что прочие вершины в этот момент будут окрашены как-то иначе, и в целом картина раскраски графа будет меняться по ходу решения задачи. Нам понадобятся всего лишь три краски, определённые константами, как описано в табл. 22-6:

Табл. 22-6 — Цвета и состояния вершин в ходе решения задачи

Цвет вершины		Состояние вершины
CWhite	**Белая**	**Вершина ещё не опробована**
CBlack	**Чёрная**	**Вершина назначена для размещения службы (претендует на роль медианной)**
CGray	**Серая**	**Вершина уже опробована в качестве медианной, и на данном уровне обработки уже не будет назначена чёрной**

Процесс окраски рассмотрим ниже, а пока достаточно понять, что в момент вызова функции, назначенные для размещения служб вершины, окрашены чёрным. Тогда в ходе подсчёта стоимости текущего «чёрного подмножества» достаточно перебрать нечёрные вершины и определить расстояния каждой из них до ближайшей чёрной. Эта работа станет совсем простой с применением карты для быстрого поиска маршрутов. Напомню, что эту распределённую по вершинам карту мы строили при поиске центров и p-центров процедурой **InitMap**. В каждой вершине графа она создаёт список дальних указателей типа **TFarLink**, где поля **mNodeFar** и **mDist** содержат соответственно ссылку на конечную вершину и расстояние между нею и данной вершиной:

```
TFarLink = class(TItem)    // Дальний указатель
    mNodeFar : TNode;      // целевая вершина
    mNodeNear: TNode;      // ближайшая вершина на пути к целевой
    mDist    : integer;    // расстояние от текущей к целевой
    . . .
  end;
```

За подробностями построения карты дальних указателей отсылаю к главам 15 и 20. Отметим, что элементы **TFarLink** упорядочены в списках по неубыванию расстояний **mDist**, а это важно для нашего алгоритма. Положим, что в момент вызова метода **MedianCost** карта дальних указателей уже построена.

Листинг 22-2 — Вычисление стоимости p-подмножества

```
// Вычисление стоимости p-подмножества
// (вершины в этот момент уже раскрашены)
// Цвета вершин означают:
// CWhite -- белые ещё не опробованы
// CBlack -- чёрные пробуются в качестве медианных
// CGray  -- серые уже побывали чёрными и не могут быть медианными

function TGraph.MedianCost: integer;
var Node: TNode;       // текущая вершина
    Dist: integer;     // расстояние к ближайшей чёрной
  //.................................................
  // Возвращает расстояние к ближайшей чёрной вершине
  function FindMinDist: integer;
  var FL: TFarLink;     // элемент дальней связи
  begin
    Result:= CInfinity;
    with Node.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if FL.mNodeFar <> Node then begin
          if FL.mNodeFar.mColor = CBlack then begin
            Result:= FL.mDist;
            Break;
          end;
        end;
        FL:= GetNext as TFarLink;
      end;
    end; // with
  end;
  //.................................................
begin  { MedianCost }

  // Перебираем вершины графа (списки расстояний),
  // (вклад в стоимость дают только белые и серые вершины)
  Result:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mColor <> CBlack then begin
      // чёрные вершины не дают вклад в стоимость
      Dist:= FindMinDist;  // минимальное расстояние для Node
      Inc(Result, Node.mValue * Dist);   // накопление результата
    end;
    Node:= NodeNext;
  end;
end;
```

22.7. Упрощённый перебор

Теперь мы готовы построить упрощённый алгоритм, перебирающий все комбинации из *p* вершин. Он раскрасит граф, подсчитает стоимость каждой комбинации, и сохранит одну из самых «дешёвых». По ходу перебора комбинаций цвет каждой вершины графа будет меняться от белого сначала к чёрному, а затем к серому так, как показано на рис. 22-4.

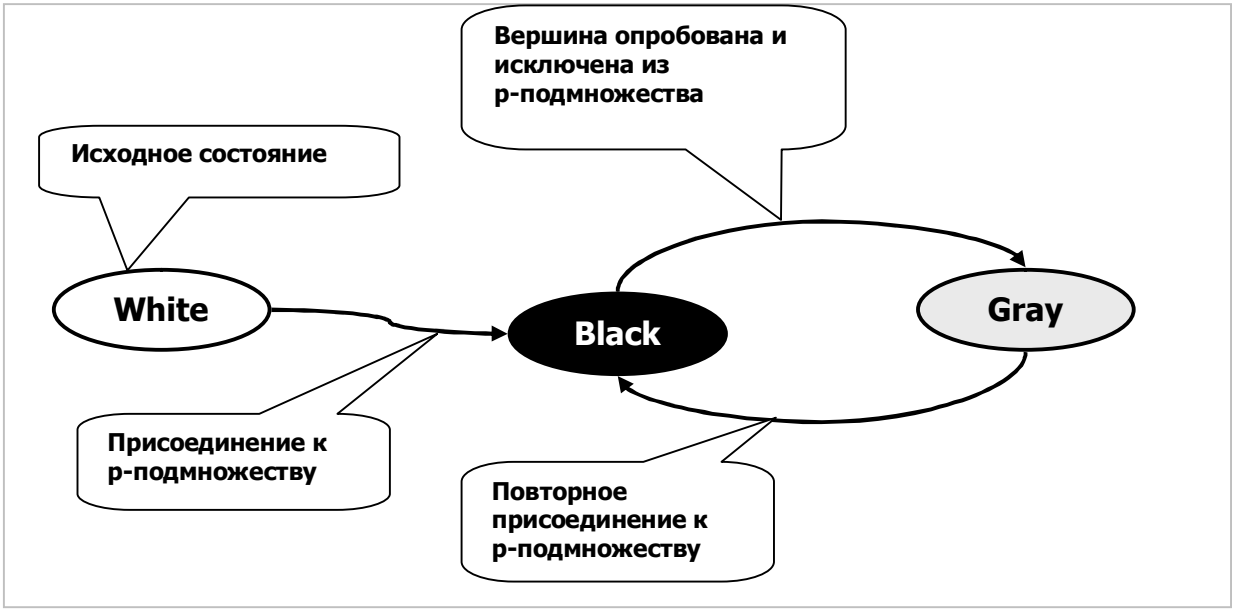


Рис. 22-4 — Диаграмма изменения цвета вершин в упрощённом алгоритме перебора

Изначально все вершины белые. По ходу перебора те из них, что включаются (неявно) в *p*-подмножество, окрашиваются чёрным. Опробовав очередное подмножество, мы удалим из него последнюю из добавленных к нему вершин — окрасим её серым, — и добавим следующую вершину на этом же уровне перебора. После обработки всех вершин текущего уровня все они станут серыми, но при повторном входе на этот уровень (из другой вершины) некоторые из них снова будут опробованы — временно окрашены чёрным.

Листинг 22-3 — Упрощённый поиск p-медианы (a-версия)

```
// Поиск медианы "в лоб" перебором
// всех комбинаций медианных подмножеств
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian_A (
                    aPoly: integer;     // кратность P
                    aDirect: TCenter;   // внешняя, внутр., внешне-внутр.
                    var aCost: integer  // стоимость (результат)
                   ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  // Рекурсивный поиск перебором всех возможных
  // комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже текущей, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут,
        // погружаемся на следующий уровень
        Inc(Level);
        Searching(i);     // рекурсивный вызов следующего уровня
        Dec(Level);
      end;
      N.mColor:= CGray;   // проверенной вершине назначаем серый цвет
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin { TGraph.GenPMedian_A }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать списки ближайших вершин (карту дальних указателей):
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // количество узлов, обраб. на каждом уровне
  Searching(1);         // начать перебор с первой вершины
  DoneMap;              // удалить множества дальних указателей
  aCost:= BestCost;     // вернуть цену
end;
```

22.8. Перебор с досрочным возвратом

Итак, задача поиска медиан, казалось бы, решена (опытную проверку доверяю читателю). Однако при значениях *P* близких к *N/2* её сложность становится экспоненциальной, а это снижает практическую ценность алгоритма. Поищем способ отсечь большую часть ветвей перебора. Пусть воображаемый лыжник, достигнув некоторого промежуточного узла перебора, хотя бы грубо, оценит

минимальную стоимость лучшего из p–подмножеств, формируемых на последующих уровнях. Если такая грубая оптимистичная оценка окажется *выше*, чем текущая оценка лучшего из уже найденных p–подмножеств, то дальнейший спуск из узла теряет смысл, и надо перейти к следующему узлу. Иначе говоря, существующие операторы:

```
// Здесь размер p-подмножества не достигнут,
// погружаемся на следующий уровень
Inc(Level);     // Level+1
Searching(i);   // рекурсивный вызов следующего уровня
Dec(Level);     // Level-1
```

надо заключить внутрь условия:

```
// Здесь размер p-подмножества не достигнут.
// Сравним оценку минимальной стоимости и лучшей текущей.
// Если она меньше текущей, то "ныряем" на уровень вниз
if  EvalBest(BestCost) then begin  // погружаемся на следующий уровень
  Inc(Level);     // Level+1
  Searching(i);   // рекурсивный вызов следующего уровня
  Dec(Level);     // Level-1
end;
```

где **`EvalBest`** — некая булева функция, дающая сравнение оптимистичной оценки с текущей лучшей оценкой. Рассмотрим принцип работы такой функции.

Отметим, что в промежуточном узле «горнолыжного» спуска чёрное подмножество сформировано лишь частично, то есть, количество чёрных вершин B ещё не достигло P (здесь $B < P$). В этот момент все вершины графа разбиты на три подмножества: W — белые, B — чёрные и G — серые, а сумма мощностей этих подмножеств равна количеству вершин графа N:

$$W + B + G = N$$

В этот момент оптимистичная оценка даётся *прикреплением* белых и серых вершин не только к ближайшим *чёрным* (как в точном подсчёте стоимости), но и к ближайшим *белым* вершинам, если в списке дальних указателей **`mFarLinks`** они окажутся ближе чёрных. Это оправдано тем, что $(P - B)$ из W белых вершин, в конце концов, станут чёрными, так почему бы не испытать их заранее? Конечно, в этот момент мы ещё не знаем, какие именно белые вершины почернеют, но это и не обязательно: мы ищем не точную, а *оптимистичную*, то есть, заведомо заниженную оценку стоимости. В ходе суммирования вклад серых вершин накапливаем сразу, а с белыми повременим. Дело в том, что к сумме надо прибавить вклад не всех, а только $(W - P + B)$ белых вершин, причём только тех, что вносят наименьший вклад в стоимость. Поэтому элементы, хранящие эти вклады, предварительно сохраним в промежуточном множестве, выстроив их по неубыванию стоимости. Затем, после обработки всех вершин графа, $(W - P + B)$ наименьших вкладов от белых вершин, взятых из промежуточного множества, добавим к сумме, а оставшиеся самые дорогие из $(P - B)$ белых вершин отбросим,

как не вносящие вклад в стоимость. Так получим оптимистичную оценку стоимости потенциально возможного и пока не сформированного p-подмножества.

Для этого расчёта надо временно изменить раскраску вершин, и потому перед возвратом на предыдущий уровень уже опробованные на данном уровне серые вершины следует вновь обелить, сделав их потенциальными кандидатами в чёрные. Новая диаграмма изменения состояния вершин станет такой (рис. 22-5):

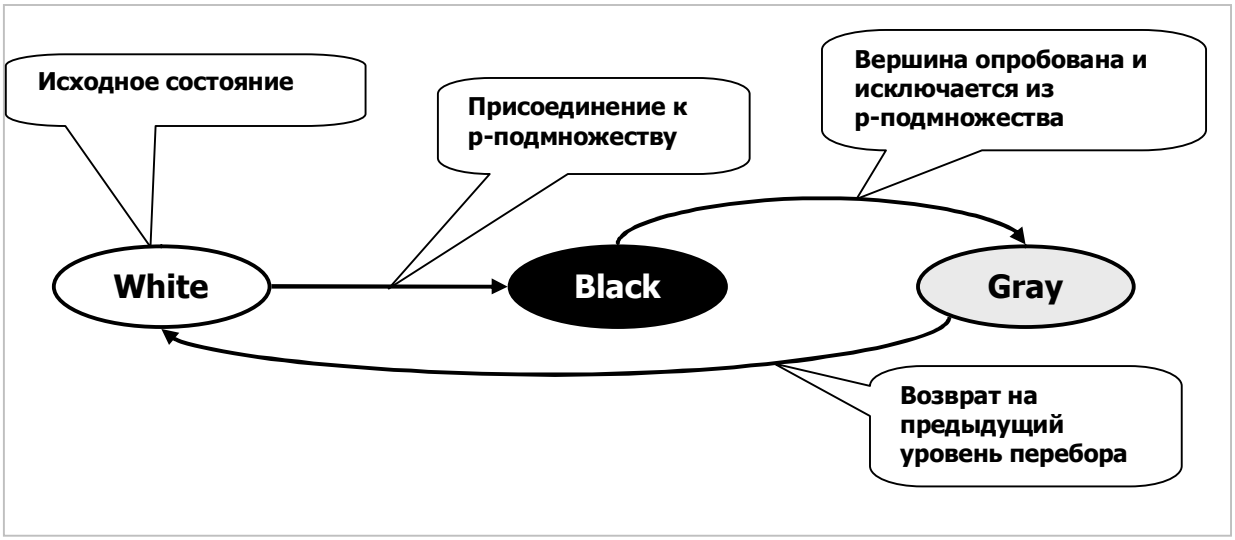


Рис. 22-5 — Диаграмма изменения цвета вершин в усовершенствованном алгоритме

Заметим, что в момент добавления очередной чёрной вершины *X* и вызова оценочной функции **`EvalBest`** все вершины, предшествующие *X*, будут либо серыми, либо чёрными (вершины перебираем в фиксированном порядке), а все последующие — белыми.

Ещё одно замечание касается параметра, передаваемого в функцию **`EvalBest`**, — это текущая лучшая оценка, мы воспользуемся ею ещё и для досрочного прекращения суммирования, когда накопленная сумма достигнет этой оценки. Усовершенствованный поиск медиан представлен в следующем листинге.

Листинг 22-4 Поиск медиан частичным перебором p-подмножеств

```
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian(aPoly: integer;   // кратность P
                           aDirect: TCenter;  // внешн., внутр., внешне-внутр.
                           var aCost: integer // стоимость (результат)
                           ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Оценка минимальной стоимости текущего распределения вершин
    // Цвета вершин означают:
    // CWhite -- белые ещё не опробованы
```

```
  // CBlack -- чёрные пробуются в качестве медианных
  // CGray  -- серые побывали чёрными и не могут быть медианными

  function EvalBest(aBest: integer): boolean;
  var Node: TNode;        // текущая вершина
        FL: TFarLink;     // элемент дальней связи
       Sum: integer;      // накопитель суммы
      Dist: integer;      // расстояние к ближайшей не серой вершине
         S: TSet;         // неубывающее множество элементов FL
      WhiteCnt: integer; // счётчик белых вершин
    //.................................................
    // Поиск ближайшей не серой вершины
    // Node -- текущий не чёрный узел
    function FindMinDist: integer;
    var FL: TFarLink;     // элемент дальней связи
    begin
      Result:= CInfinity;
      with Node.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          if (FL.mNodeFar <> Node) and
             (FL.mNodeFar.mColor <> CGray) then begin
            // Можно прикреплять к чёрным и белым
            Result:= FL.mDist;
            Break;
          end;
          FL:= GetNext as TFarLink;
        end;
      end; // with
    end;
    //.................................................

  begin  // Eval
    // Перебираем вершины графа (матрицу расстояний),
    // определяем минимальное расстояние к не серым вершинам,
    // и формируем список из неубывающих чисел (расстояний)
    // (вклад в стоимость дают только белые и серые вершины)

    Sum:= 0;  // накопитель суммы
    // начальное значение счётчика суммируемых белых вершин
    WhiteCnt:= mNodes.GetCount - aPoly;
    S:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor <> CBlack then begin
        // чёрные не дают вклад в стоимость
        Dist:= FindMinDist;  // минимальное расстояние для Node
        case Node.mColor of
          CWhite: begin
                    // взвешенное расстояние к белой вершине
                    // вставляем в неубывающий список
                    FL:= TFarLink.Create(nil, Node, Node.mValue*Dist);
                    S.Insert(FL);
                  end;
          CGray:  begin
                    // серую суммируем и уменьшаем счётчик белых
                    Inc(Sum, Node.mValue*Dist); // накопление результата
                    Dec(WhiteCnt);     // счётчик суммируемых вершин -1
                    if Sum>=aBest then Break;
                  end;
        end;
      end;
```

```
        Node:= NodeNext;
      end;
      // По окончании формирования множества расстояний
      // добавляем WhiteCnt первых из них к накопленному результату
      FL:= S.GetFirst as TFarLink;
      while (WhiteCnt > 0) and (Sum < aBest) do begin
        Inc(Sum, FL.mDist);           // накопление суммы
        FL:= S.GetNext as TFarLink;   // следующий элемент списка
        Dec(WhiteCnt)
      end;
      // Очистка и удаление временного списка:
      S.ClrAndDestroy;
      S.Free;
      Result:= Sum < aBest;
    end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный поиск перебором части комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже текущей, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут.
        // Сравним оценку минимальной стоимости с лучшей текущей.
        // Если она меньше текущей, то "ныряем" на уровень вниз
        if  EvalBest(BestCost) then begin
          // погружаемся на следующий уровень
          Inc(Level);    // Level+1
          Searching(i);  // рекурсивный вызов следующего уровня
          Dec(Level);    // Level-1
        end;
      end;
      N.mColor:= CGray;  // проверенной вершине назначаем серый цвет
    end;
    // Перед возвратом на уровень вверх возвращаем белый цвет
    // всем обработанным вершинам текущего уровня
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CWhite;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.GenPMedian }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
```

```
  // Сформировать множества ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // кол-во узлов, обраб. на каждом уровне
  Searching(1);         // начать перебор с первой вершины
  DoneMap;              // удалить множества дальних указателей
  aCost:= BestCost;     // вернуть цену
end;
```

22.9. Испытания

Для испытания метода поиска медиан применена следующая программ:

Листинг 22-5 — Программа для испытания метода поиска медиан

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;           // граф
    Median: TSet;          // медиана
    Cost: integer;         // стоимость медианы
    Direct : TCenter;      // направление медианы
    P : integer;           // кратность
    D : char;              // для ввода данных о направлении
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    Median:= Gr.GenPMedian (P, Direct, Cost) ;
    Median.Expo;
    Median.Free;
    Writeln('Cost  = ', Cost:7);
  until false;
end.
```

«Подопытный» граф представлен следующим текстом и показан на рис. 22-6, а результаты поиска его p-медиан даны в табл. 22-7.

```
Test.txt – орграф с взвешенными дугами и вершинами
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

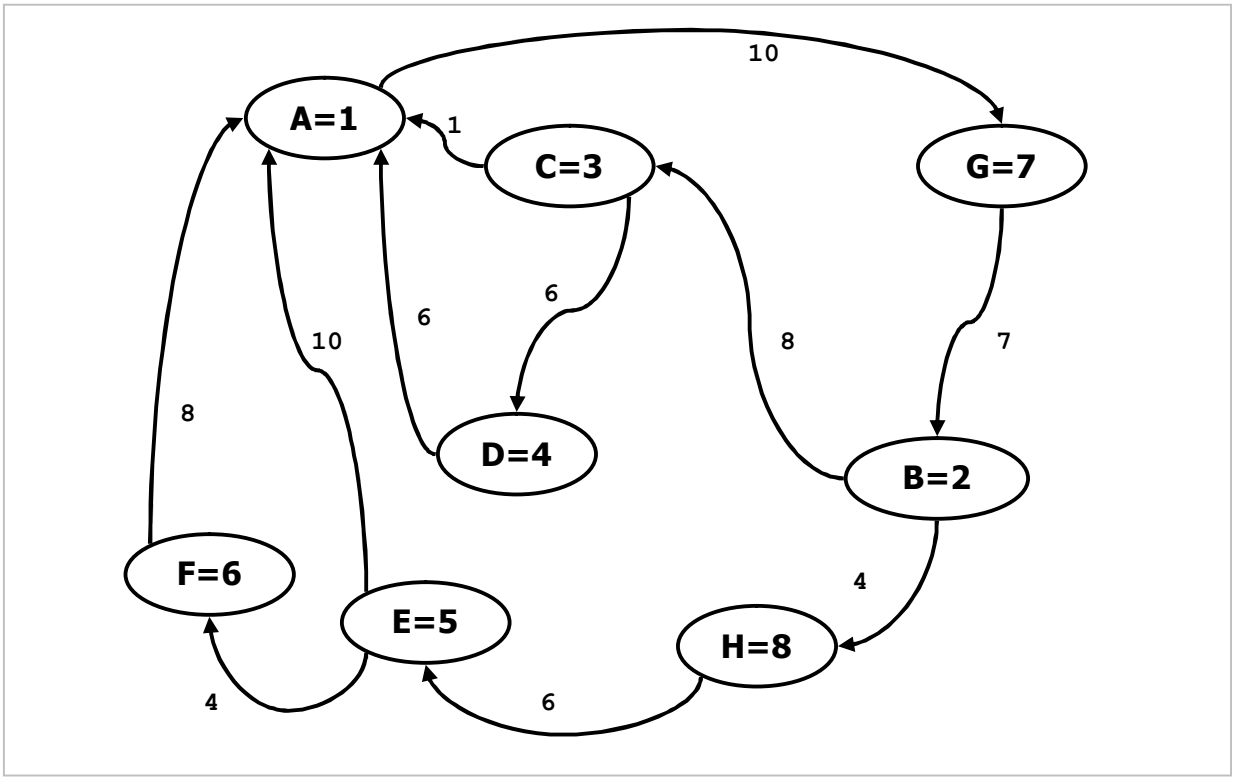


Рис. 22-6 — Орграф с взвешенными дугами и вершинами

Табл. 22-7 — P-медианы графа, показанного на рис. 22-6

Кратность P	Внешние (Out)		Внутренние (In)		Внешне-внутренние (In-Out)	
	Вершины	Стоимость	Вершины	Стоимость	Вершины	Стоимость
1	B	388	A	383	G	1019
2	C, H	228	A, H	210	G, H	723
3	C, G, H	129	A, F, H	132	F, G, H	489
4	C, E, G, H	63	A, F, G, H	55	E, F, G, H	304
5	C, D, E, G, H	39	A, D, F, G, H	31	D, E, F, G, H	156

Следующая программа сравнивает быстродействия двух методов — полного и частичного перебора:

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;           // граф
    Median: TSet;          // медиана
    Cost: integer;         // стоимость медианы
    Direct : TCenter;      // направление медианы
    P : integer;           // кратность
    D : char;              // для ввода данных о направлении

procedure TestTime;
const C1 =20; C2 =5; // количество повторений
var i: integer;
    Start, Time : TDateTime;
begin

  case Direct of
    InCenter: Writeln('------------------- IN -------------------');
    OutCenter: Writeln('------------------- OUT -------------------');
    InOutCenter: Writeln('------------------- IN-OUT -------------------');
  end;
  Start:= Now;
  for i:= 1 to C1 do begin
    Median:= Gr.GenPMedian (P, Direct, Cost);
    if i < C1 then Median.Free;
  end;
  Time:= MilliSecondsBetween(Start, Now) / C1;
  Median.Free;
  Writeln('Cost  = ', Cost:7, '  Time  = ', Time:7:2);

  Start:= Now;
  for i:= 1 to C2 do begin
    Median:= Gr.GenPMedian_A (P, Direct, Cost);
    if i < C2 then Median.Free;
  end;
  Time:= MilliSecondsBetween(Start, Now) / C2;
  Median.Free;
  Writeln('Cost_A= ', Cost:7, '  Time_A= ', Time:7:2);
end;

begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('- - - - - - - - - - - - - - - - - - -');
  Write('Direction (I/O/A) = '); Readln(D);
  case Upcase(D) of
    'I': Direct:= InCenter;
    'O': Direct:= OutCenter;
    else Direct:= InOutCenter;
  end;
```

```
  repeat
    Write('P= '); Readln(P);
    if P=0 then Break;
    TestTime;
    Writeln;
  until false;
end.
```

Для обработки взят граф из 24 вершин, его текстовое представление дано ниже:

```
Test.txt
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
24 - количество вершин
A=9 B=3 C=3 D=7 E=4 F=2 G=4 H=5 I=1 J=5 K=1 L=9 M=1 N=3 O=10 P=4 Q=8 R=4 S=7
T=9 U=8 V=4 W=2 X=4
A -> E=4 H=10 I=8 O=8 T=2
B -> C=8 E=7 H=2 L=9 R=1
C -> B=8 D=1 H=8 J=1 O=7 Q=5
D -> C=1 E=10 F=10 G=6 I=10 L=9 M=1 O=8 R=3 X=1
E -> A=4 B=7 D=10 G=6 J=9 K=7 N=7 S=10 X=2
F -> D=10 K=2 L=9 N=3 X=7
G -> D=6 E=6 H=8 I=6 L=2 P=9 Q=1 U=8 W=8
H -> A=10 B=2 C=8 G=8 M=5 P=7 S=8 V=5
I -> A=8 D=10 G=6 J=1 O=8 T=9 U=4 V=4
J -> C=1 E=9 I=1 T=7
K -> E=7 F=2 M=5 R=5
L -> B=9 D=9 F=9 G=2 R=7 V=6 X=9
M -> D=1 H=5 K=5 Q=1 W=4
N -> E=7 F=3 O=9 Q=1 R=5 U=7 W=7
O -> A=8 C=7 D=8 I=8 N=9 S=9 U=3 W=7
P -> G=9 H=7 S=8
Q -> C=5 G=1 M=1 N=1 T=6 X=7
R -> B=1 D=3 K=5 L=7 N=5 V=5
S -> E=10 H=8 O=9 P=8 U=10 X=8
T -> A=2 I=9 J=7 Q=6
U -> G=8 I=4 N=7 O=3 S=10 W=7
V -> H=5 I=4 L=6 R=5
W -> G=8 M=4 N=7 O=7 U=7
X -> D=1 E=2 F=7 L=9 Q=7 S=8
```

Результаты сравнения двух методов даны в табл. 22-8 и представлены на графике рис. 22-7.

Табл. 22-8 — Время поиска p-медиан частичным (T1) и полным (T2) перебором, мс

Кратность медианы P	Частичный перебор (T1)	Полный перебор (T2)	Кратность медианы P	Частичный перебор (T1)	Полный перебор (T2)
1	5,5	6,2	13	18,8	5534,0
2	6,2	6,0	14	12,5	4260,0
3	14,0	15,6	15	8,6	2838,0
4	39,0	43,8	16	7,0	1597,0
5	57,0	147,0	17	7,0	762,0
6	65,0	419,0	18	6,2	312,0
7	69,0	988,0	19	6,2	106,0
8	62,0	1975,0	20	6,2	34,2
9	55,0	3344,0	21	6,2	12,2
10	40,0	4810,0	22	5,5	6,2
11	30,0	6010,0	23	5,5	6,2
12	23,5	6225,0			

Рис. 22-7 — Время поиска p-медиан в зависимости от кратности P

Как и следовало ожидать, время поиска p-медиан перебором всех подмножеств подчинено «колокольному» биномиальному закону и многократно возрастает с приближением *P* к значениям, близким к *N/2*. В частичном переборе эта зависимость выражена слабо, и в целом этот более сложный алгоритм оправдывает себя, многократно обгоняя «тупой» перебор.

22.10. Итоги

22.10.1. Медианой графа называют вершину, для которой сумма взвешенных расстояний между нею и другими вершинами минимальна. Для ориентированных графов различают три рода медиан: внешнюю, внутреннюю и внешне-внутреннюю.

22.10.2. Кратной P-медианой графа называют подмножество из P вершин, для которого сумма взвешенных расстояний между этими вершинами и прочими вершинами графа минимальна. В орграфах различают три рода P-медиан: внешнюю, внутреннюю и внешне-внутреннюю.

22.10.3. Для поиска медиан и P-медиан применяется один алгоритм, упрощённая версия которого состоит в полном переборе всех возможных p–подмножеств и выборе самого «дешёвого».

22.10.4. Перебор всех p–подмножеств при значениях P, близких к N/2 приобретает экспоненциальную сложность, и потому непригоден для больших значений N (от нескольких десятков и более).

22.10.5. Предварительная оценка нижней границы стоимости потенциальных p–подмножеств, выполняемая в промежуточных точках перебора, позволяет отбросить большую часть ветвей и ускорить поиск на много порядков.

22.11. Задания

22.11.1. Учёт веса вершин

Модифицируйте поиск медиан с учётом различной стоимости постройки служб в разных вершинах графа. Добавьте необходимое поле в структуру графа и модифицируйте алгоритмы подсчёта окончательной стоимости и предварительной оценки.

22.12. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 23
Остовные деревья

23.1. Неориентированные деревья

Недавно у администрации архипелага появились (наконец-то!) средства на замену изношенных мостов. Из экономии новые двунаправленные мосты решили строить на месте старых так, чтобы при минимальной их общей длине сохранилась взаимная доступность островов (все старые мосты демонтировали). Новая мостовая схема образовала *остов* или *покрывающее дерево* графа (в данном случае это дерево минимально). На рис. 23-1 представлен один из минимальных остовов архипелага (старые мосты показаны пунктиром):

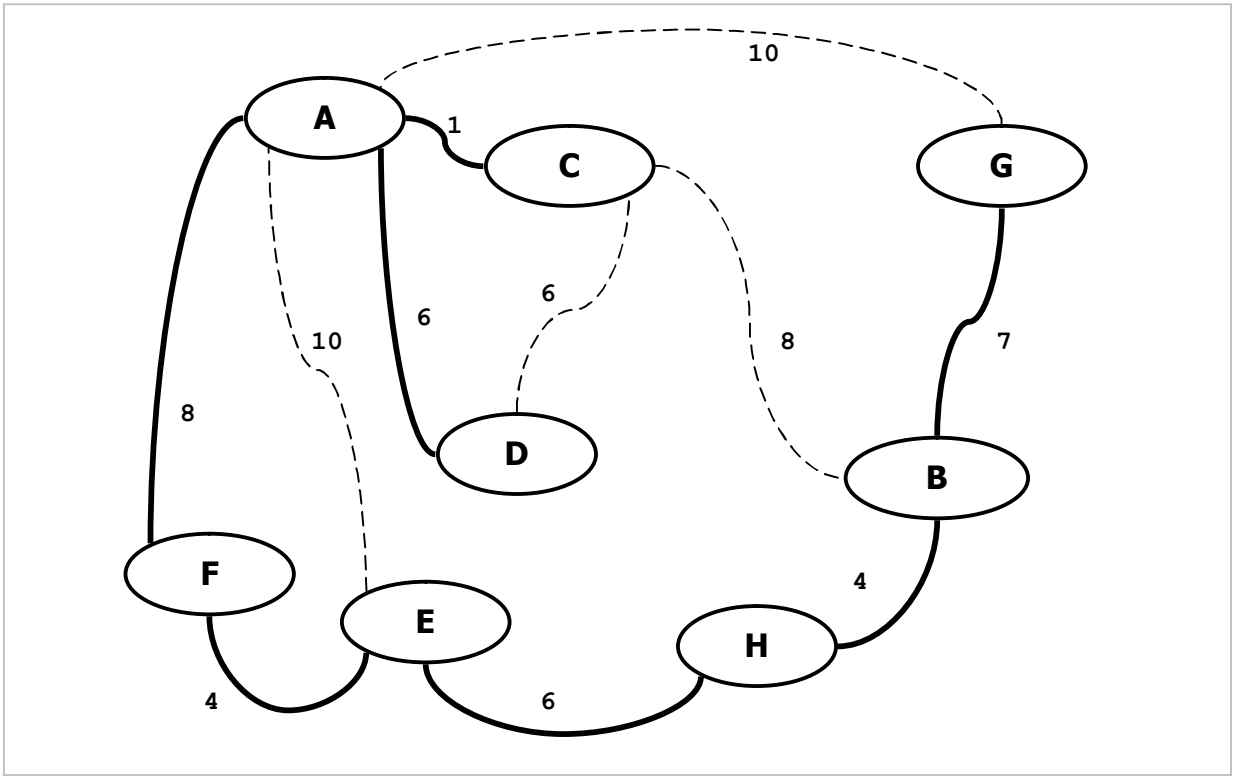


Рис. 23-1 — Один из двух минимальных остовов графа

Это остовное дерево построено на рёбрах, оно не ориентировано. Скоро нам понадобятся свойства *дерева*, и потому дадим три эквивалентных его определения (любые два из них следуют из третьего):

- дерево — это связанный граф, состоящий из N вершин и $N-1$ рёбер;
- дерево — это связанный граф, не имеющий циклов;
- дерево — это граф, в котором каждая пара вершин связана одной и только одной простой цепью.

В самом деле, если предположить, что в новой мостовой схеме существует цикл, то некоторые острова свяжутся двумя цепочками, — стало быть, часть из мостов избыточна, — такой граф не будет деревом.

Наряду с *минимальным* остовом, на графе можно построить и другие деревья, подпадающие под наши определения (например, остов *максимального* веса). При решении иных проблем надо перебирать *все* или *часть* остовных деревьев графа, поэтому ниже мы рассмотрим две задачи:

- построение остовов *минимального* или *максимального* веса;
- перебор *всех* остовных деревьев графа.

Алгоритмы построения остовных деревьев используют свойства дерева и оперируют с рёбрами графа или дугами орграфа.

23.2. Минимальный (максимальный) остов — алгоритм Прима

Принцип построения *минимального* (или *максимального*) остова крайне прост, и основан на равноправии вершин неориентированного графа. Поскольку вершины равноправны, начать постройку дерева можно с любой из них, например, с первой по порядку (с вершины *A*). Взяв её и окрасив чёрным (а изначально все вершины белые), мы отмечаем её, как обработанную. Далее находим среди белых вершин *ближайшую* к ней, пусть это будет вершина *X*, красим её чёрным, а ребро *A-X* включаем в состав дерева. На следующем шаге среди белых вершин ищем ближайшую к «чёрному» подмножеству, то есть, либо к вершине *A*, либо к вершине *X*. Пусть это будет вершина *Y*, тогда *Y* очерняем, а кратчайшее из рёбер (*A-Y* или *X-Y*) присоединяем к дереву. Этот процесс продолжаем вплоть до почернения всех вершин графа. Максимальный остов строится аналогично, но выбираются *длиннейшие* рёбра.

Представленный тут алгоритм *Прима* слегка похож на расширение гаммы при поиске кратчайших путей алгоритмом Дейкстры. Но если там мы расширяли чёрное подмножество включением всех соседей, то здесь на каждом шаге присоединяем только одного из ближайших. Это обстоятельство вынуждает нас позаботиться о быстром доступе к ближайшему белому соседу, для чего потребуется особый объект. Этот объект нужен ещё и потому, что в неориентированном графе рёбра представлены парами встречно направленных дуг (парами линков **TLink**). Но в решении задачи для нужд перебора каждую такую пару надо представить одним объектом. Итак, на основе класса **TLink** сконструируем новый вспомогательный линк:

```
TTreeLink = class (TLink)
  mMaxTree: boolean;  // false -- минимальный, true - максимальный остов
  constructor Create(aLink: TLink);
  function  Compare(arg: TItem): TCompare; override;
  procedure Print(var aFile: TextFile); override;
end;
```

Вспомогательные линки соответствуют реальным рёбрам, хоть и не привязаны к вершинам графа. Своим методом сравнения **Compare** эти линки обеспечат нам и нужный порядок сортировки связей по неубыванию длины, и отсечение возможных дубликатов:

Листинг 23-1 — Метод сравнения линков, устраняющий дубликаты

```
function TTreeLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Для неориентированного графа сравниваем исходную и конечную вершины
  // на предмет встречных линков
  if not mOwner.mOwner.mDirect and  // если не орграф
    (mOwner=(arg as TTreeLink).mDest) and (mDest=(arg as TTreeLink).mOwner)
    then begin
      // Здесь линки направлены встречно, отвергаем дубликат:
      Result:= cmpEq;
      Exit;
    end;
  // Если линки не совпадают, то сортируем по неубыванию веса (длины)
  Result:= cmpLess;
  if mValue > (arg as TTreeLink).mValue then Result:= cmpGreate
  // Для построения дерева максимального веса результат инвертируем
  if mMaxTree
    then if Result=cmpLess then Result:= cmpGreate else Result:= cmpLess;
 end;
```

Напомню, что когда метод **Compare** возвращает значение **cmpEq**, метод **TSet.Insert** не вставляет элемент в множество и возвращает **false**.

Порождение вспомогательных линков для некоторой вершины поручим методу этой вершины **TNode.AddTreeLinks**, куда накопитель линков передаётся через параметр:

Листинг 23-2 — Добавление линков, ведущих к ближайшим белым вершинам

```
// aLinks - накопитель (буфер) линков
// Линки выстраиваются по неубыванию (aMaxTree=false)
// или невозрастанию дистанции (aMaxTree=true)

procedure TNode.AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
var  L : TLink;
    TL : TTreeLink;
begin
  // Обработка исходящих связей
  L:= OutLinkFirst;
  while Assigned(L) do begin
    // Вставляем только линки, ведущие к белым вершинам
    if (L.mDest.mColor = CWhite) then begin
      TL:= TTreeLink.Create(L, aMaxTree);
      if not aLinks.Insert(TL) then TL.Free;
    end;
    L:= OutLinkNext;
  end;
end;
```

Здесь при попытке вставить дубликат метод **`Insert`** возвращает **`false`**, и тогда дубликат уничтожается.

Разобравшись с формированием вспомогательных линков, рассмотрим метод, формирующий остов минимального или максимального веса (алгоритм Прима):

Листинг 23-3 — Построение минимального остовного дерева методом Прима

```
function TGraph.GenCoverTree(aMaxTree: boolean;  // false= min, true = max
                             var aCost: integer  // возвр. общую стоимость
                            ): TSet;             // множество TTreeLink
var Node: TNode;       // текущая вершина
      TL: TTreeLink;   // текущий линк для дерева
   Links: TSet;        // вспомогательное множество линков
begin
  aCost:=0;              // накопитель суммы
  Result:= CreateSet;    // множество линков для результата
  Links:= CreateSet;     // множество линков для буфера
  ResetNodes;            // очистка для всех вершин mColor = CWhite
  Node:= NodeFirst;      // исходная вершина
  // Присоединение всех вершин графа (линков на единицу меньше)
  while Assigned(Node) and
       (Result.GetCount < mNodes.GetCount-1) do  begin
    Node.mColor:= CBlack;     // присоединяем текущую к множеству чёрных
    // Добавляем в множество Links линки к ближайшим белым вершинам:
    Node.AddTreeLinks(aMaxTree, Links);
    // Перебираем линки Links в поиске ближайшей белой вершины
    TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
    Node:= nil;                         // искомая вершина пока пуста
    while Assigned(TL) do begin
      Links.Delete(TL);                 // удаляем линк из буфера Links
      // Искомая вершина может быть как источником,
      // так и приёмником связи:
      if TL.mOwner.mColor = CWhite
        then Node:= TL.mOwner   // источник связи
      else if TL.mDest.mColor = CWhite
        then Node:= TL.mDest;   // приёмник связи
      if Assigned(Node) then begin
        // Здесь ближайшая белая вершина найдена:
        Result.Insert(TL);      // вставляем линк в результат
        Inc(aCost, TL.mValue);  // наращиваем стоимость
        Break;
      end;
      // здесь линк связывает две чёрные вершины:
      TL.Free;                          // удаляем ненужный линк
      TL:= Links.GetNext as TTreeLink; // и берём следующий
    end;
  end;
  // Если остов не построен, то граф не связан
  if Result.GetCount < mNodes.GetCount-1 then begin
    Result.ClrAndDestroy;
    aCost:=0;
  end;
  // Очистка и удаление вспомогательного множества линков:
  Links.ClrAndDestroy;
  Links.Free;
end;
```

Выбрав первую вершину и окрасив её чёрным, вставляем исходящие из неё линки во вспомогательное множество **`Links`**, причём первый линк в нём будет кратчайшим (строим остов минимального веса). Следующий далее вызов **`Links.GetFirst`** выберет этот кратчайший линк, после чего линк из буфера надо удалить и проверить цвета его исходящей и входящей вершин. Если одна из них окажется белой, то линк сохраним в множестве результата (присоединим к остову), а белую вершину окрасим чёрным. Если же линк связывает две чёрные вершины (порождает цикл), отвергнем линк и извлечём из буфера следующий.

В ходе построения остова множество порождаемых линков «щетинится», подобно ёжу. На рис. 23-2 — рис. 23-5 показаны состояния вершин и линков в ходе обработки первых четырёх вершин. Обратите внимание: после присоединения вершины *C* в начале очереди окажутся два линка с одинаковой длиной: *A-D* и *C-D*. Если первым из буфера буден извлечён *A-D*, то к моменту обработки *C-D* вершина *D* станет чёрной, и тогда линк *C-D* будет отвергнут (рис. 23-5). Конечный результат построения минимального остова этого графа дан на рис. 23-1.

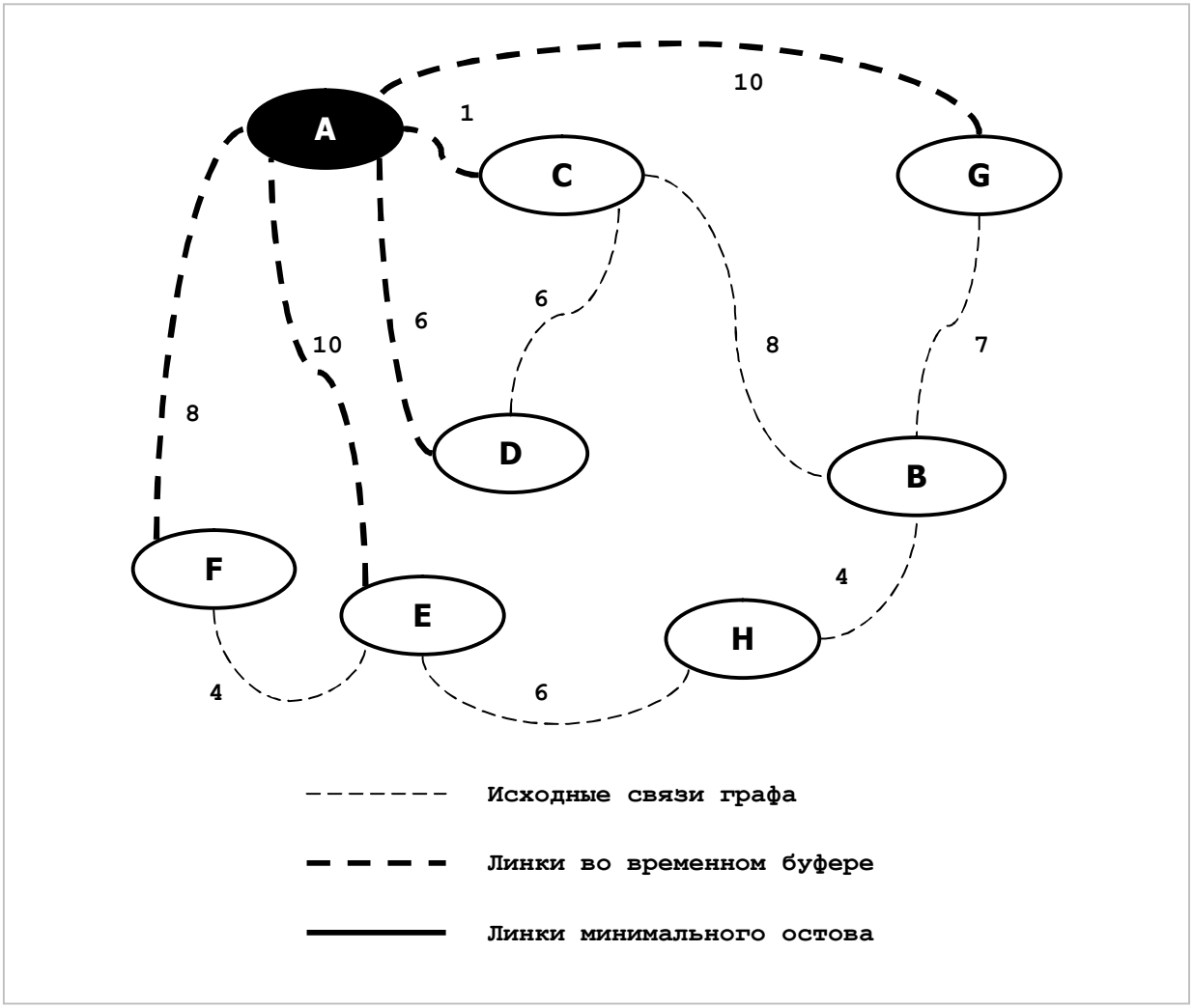


Рис. 23-2— Состояние вершин и линков после обработки 1-й вершины

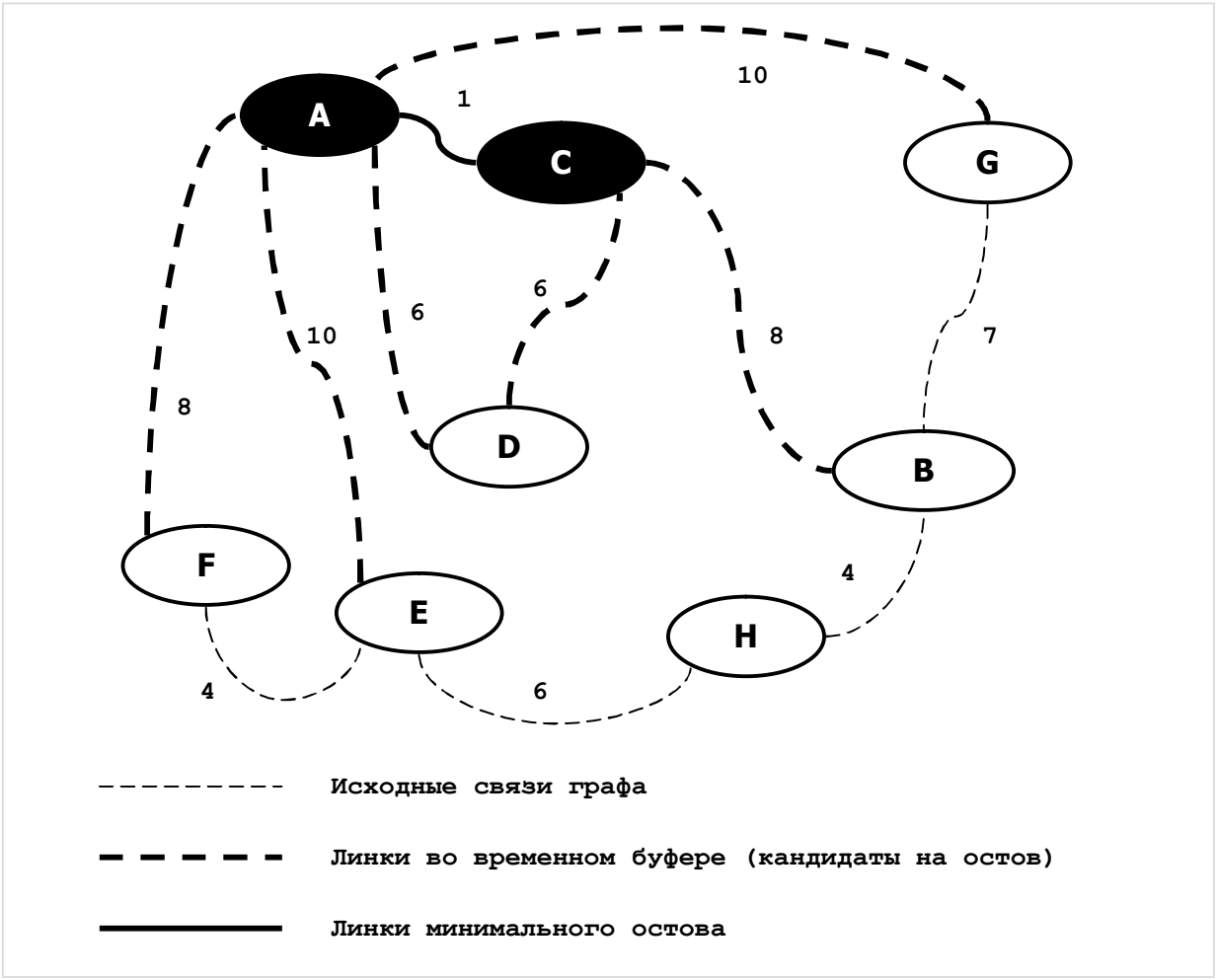


Рис. 23-3 — Состояние вершин и линков после обработки 2-й вершины

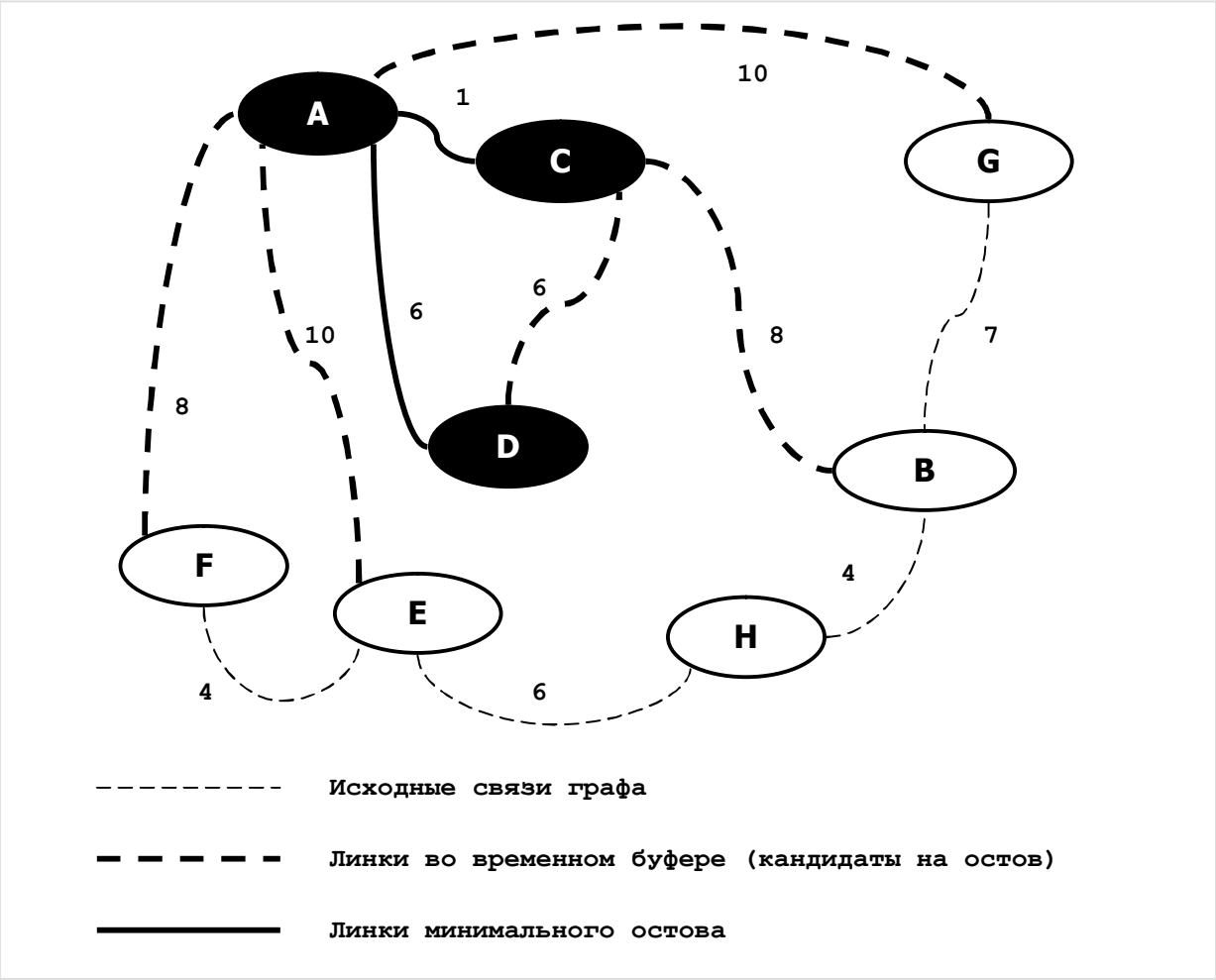


Рис. 23-4— Состояние вершин и линков после обработки 3-й вершины

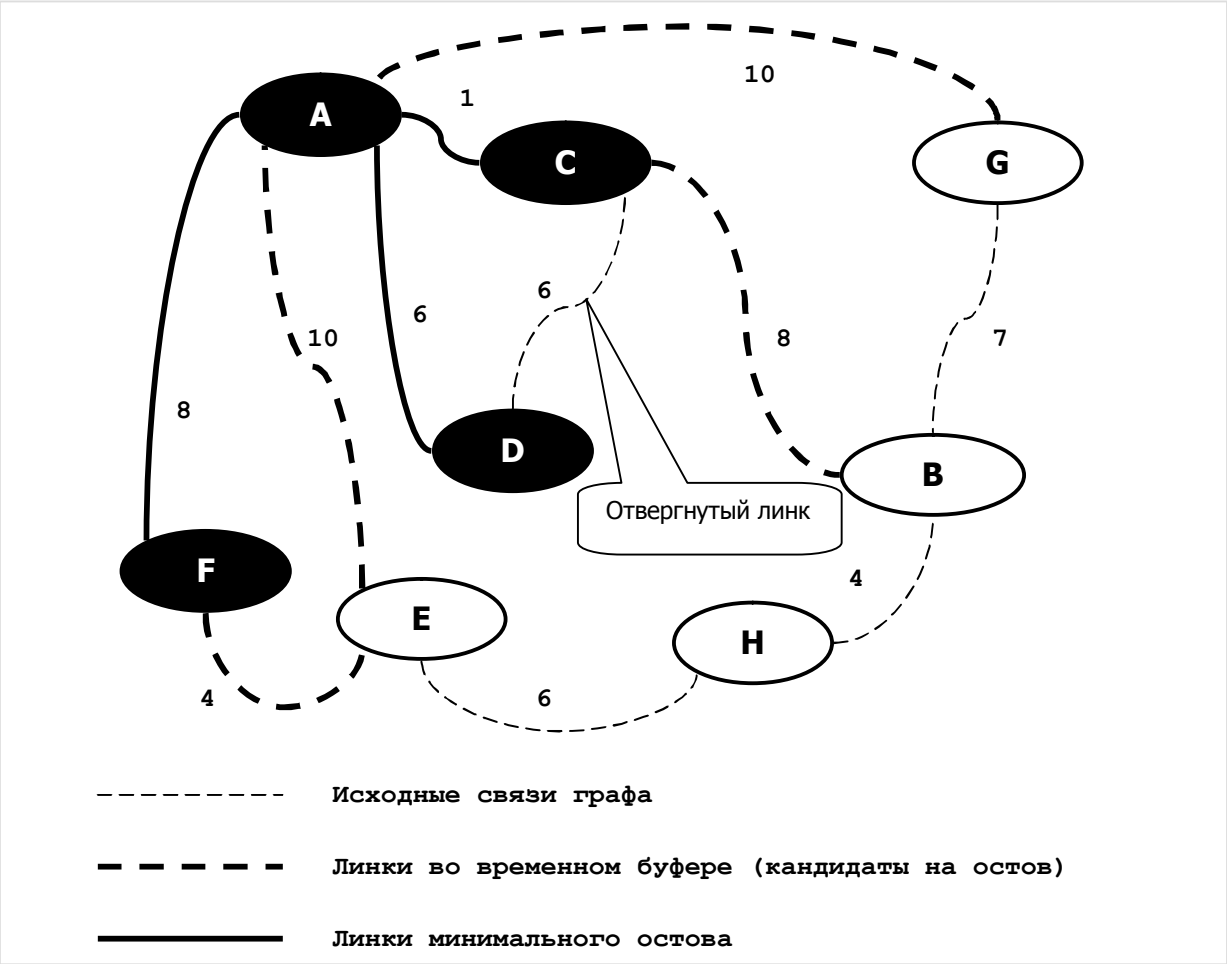


Рис. 23-5 — Состояние вершин и линков после обработки 4-й вершины

23.3. Построение всех остовов неориентированного графа

Некоторые задачи требуют перебора *всех* или *части* остовных деревьев, вот пример. Пусть на острове *X* добывают уголь, а на *Y* — руду, и для нужд развитой там металлургии перевозят уголь и руду встречными составами, — чтобы не гонять порожняк. Пусть также на острове *Z* (или на нескольких островах) устроены курорты и потому движение товарных составов там нежелательно. Тогда в связи с постройкой новых мостов можно поставить дополнительное условие: найти минимальный остов, в котором путь между вершинами *X* и *Y* не будет пролегать через курортные зоны. Таких задач, где выбор подходящих остовов специфичен, немало, и потому мы построим здесь лишь метод перебора всех остовов, вынося их обработку за скобки. Метод может стать основой многих специальных алгоритмов.

23.3.1. Оценка сложности

Когда речь заходит о переборе, первым делом хочется оценить вычислительную сложность задачи. Поскольку любой остов содержит *N-1* рёбер, где *N* — количество вершин графа, то для графа с *L* связями количество возможных комбинаций составит:

$$C_{N-1}^{L}$$

С ростом числа рёбер L количество комбинаций растёт экспоненциально. И хотя многие из них не составят деревьев, годных тоже будет немало. Доказано, что точное количество остовных деревьев вычисляется через определитель матрицы, получаемой в результате перемножения матрицы инциденций графа на её транспонированную копию. Почему бы вам не проверить это? Но мы, как сказал полководец, «пришли их бить, а не считать», и потому займёмся не подсчётом, а генерацией всех остовных деревьев. Обсуждаемые ниже алгоритмы и процедуры будут входить в состав метода **ExpoAllTrees**, формирующего и выводящего на экран все остовы графа.

23.3.2. Инициализация

Поскольку все возможные остовы порождаются перебором комбинаций из L связей, вначале позаботимся о «сырье», необходимом для этой работы, то есть подготовим множество вспомогательных линков, описанных ранее при построении минимального остова. Эта простую работу выполним при инициализации, в результате чего будет сформировано множество **Links**, упорядоченное в порядке не убывания длины линков (линки соотносятся с рёбрами исходного графа). Здесь же инициируются поля вершин, о назначении которых будет сказано позднее.

Листинг 23-4 — Инициализация вершин и построение множества линков

```
procedure Local_Init;
var Node: TNode;   // текущая вершина
begin
  ResetNodes;          // для всех вершин mColor=0, mPred=nil
  Tree:= CreateSet;    // множество для остовного дерева
  Links:= CreateSet;   // множество для всех линков
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      mRoot:= Node;                // корень поддерева
      AddTreeLinks(false, Links);  // добавление линков вершины
    end;
    Node:= NodeNext;
  end;
end;
```

23.3.3. Проверка на цикл

Вначале, пока множество линков остовного дерева **Tree** ещё пусто, вершины графа трактуются как множество изолированных поддеревьев будущего остовного дерева (лес), — эти вершины пока логически не связаны, изолированы (их связи в исходном графе тут не в счёт). В ходе построения остова будем постепенно «соединять» эти вершины линками, извлекаемыми из множества **Links**, комбинируя их всеми возможными способами, но это «соединение» будет

воображаемым, виртуальным. Логическую связь между вершинами организуем через поле **mPred**, которое применялось ранее для поиска кратчайших путей, детали обсудим ниже.

В ходе сращивания поддеревьев (логического соединения вершин) надо проверять очередной извлекаемый из множества линк на предмет того, объединяет ли он *два* поддерева, или связывает две вершины *одного*, замыкая тем самым не нужный нам цикл. Если обе вершины линка принадлежат *одному* поддереву, линк надо отбросить. В алгоритме Прима для такой проверки мы окрашивали вершины двумя цветами. Но сейчас соединяемых частей будет много, и потому для проверки на цикл в вершину введено поле **mRoot**, — специальная метка, указывающая на корень поддерева. Этот корень будет един для всех вершин одного поддерева, но в разных поддеревьях корни будут разными. Корнем может быть любая вершина поддерева, после инициализации **mRoot** указывает на саму эту вершину.

Роль *корня* подобна роли вожака в стае животных. Одиночка сам себе вожак, но в паре главенствует один. По мере слияния стай вожаком становится сильнейший из лидеров. При разделении стаи в отделившейся группе неминуемо появится свой вожак (не обязательно бывший). Здесь на «вожака стаи» показывает поле **mRoot** — указатель на одну из вершин поддерева, а порядок в «стае» поддерживается полем **mPred**, причём так, что все вспомогательные связи направляются от периферии к корню. Тогда для проверки того, что линк не создаёт цикла и объединяет два поддерева, достаточно сравнить поля **mRoot** в вершине-источнике и вершине-приёмнике линка:

```
Result:= mOwner.mRoot <> mDest.mRoot; // true, если линк не образует цикла
```

23.3.4. Сращивание деревьев

Напомню, что основой будущего алгоритма будет перебор без повторений комбинаций из *N-1* линков. А этот процесс идёт попеременно в двух направлениях. *Спускаясь* по дереву перебора, мы будем добавлять линки к множеству, формируя постепенно очередной остов, а возвращаясь *вверх*, будем откатываться в предыдущие состояния с тем, чтобы опробовать присоединение других линков. По мере спуска отдельные поддеревья леса *сращиваются*, а при подъёме вновь *расщепляются* удалением последних присоединённых линков. И потому нам потребуются две процедуры: для *сращивания*, и для *расщепления* поддеревьев.

Обратимся к рис. 23-6, где показаны два поддерева, уже порождённые на предыдущих этапах спуска.

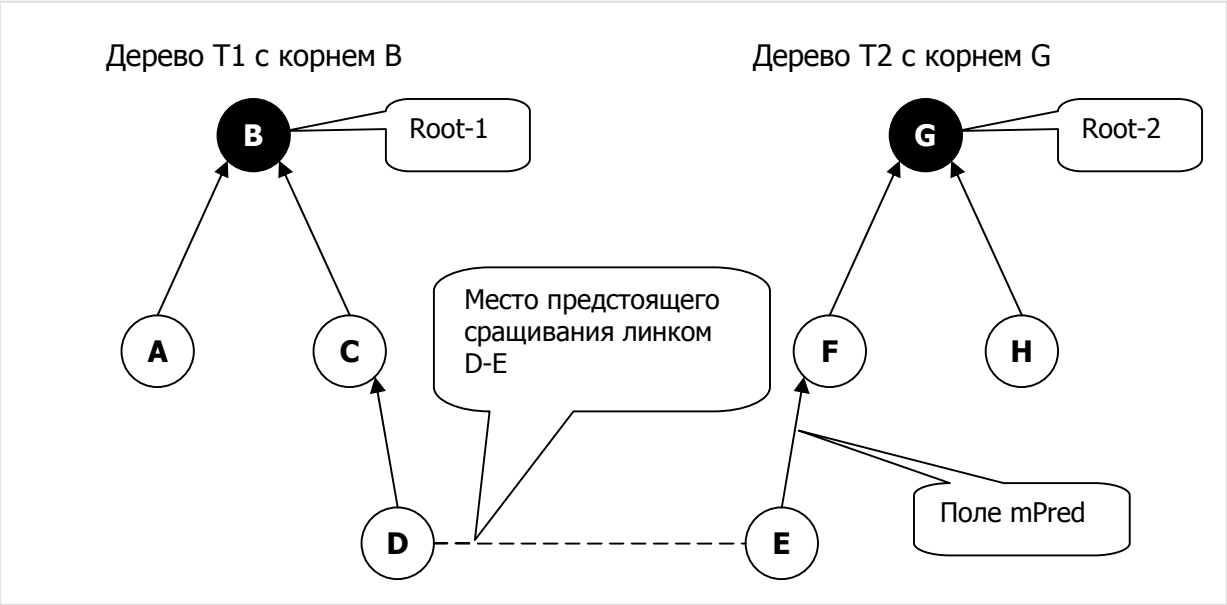


Рис. 23-6 — Исходное состояние перед сращиванием поддеревьев

Стрелки, исходящие из вершин, образованы полями **`mPred`** и организованы в ходе предшествующего сращивания. Каждой такой стрелке соответствует линк — ребро исходного графа. Вершины, не содержащие исходящих стрелок, это «вожаки стаи» — корни поддеревьев. Все вершины в дереве *T1* помечены корнем *B* (поле **`mRoot`**), а в дереве *T2* — соответственно корнем *G*.

Пусть в некоторый момент для опробования выбран линк *D-E*. Поскольку он соединяет два разных поддерева, то цикл нам не грозит, и две «стаи» соединяются. Здесь надо выбрать общего «вожака» — один из двух корней, а затем исправить некоторые стрелки так, чтобы они развернулись к этому новому корню. Который из двух корней предпочтительней? Всё равно, оба они равноправны, но для определённости выберем тот, что является корнем для источника (**`mOwner`**) обрабатываемого линка. Поскольку источник линка *D-E* — вершина *D*, то общим корнем назначим вершину *B* (см. рис. 23-6).

Выбрав корень, мы должны отразить все изменения только в подчинённом поддереве *T2*, а именно: исправить поле **`mRoot`** и развернуть стрелки, которые пока не ведут к новому корню. Такие стрелки расположены между местом состыковки *E* и старым корнем *G*.

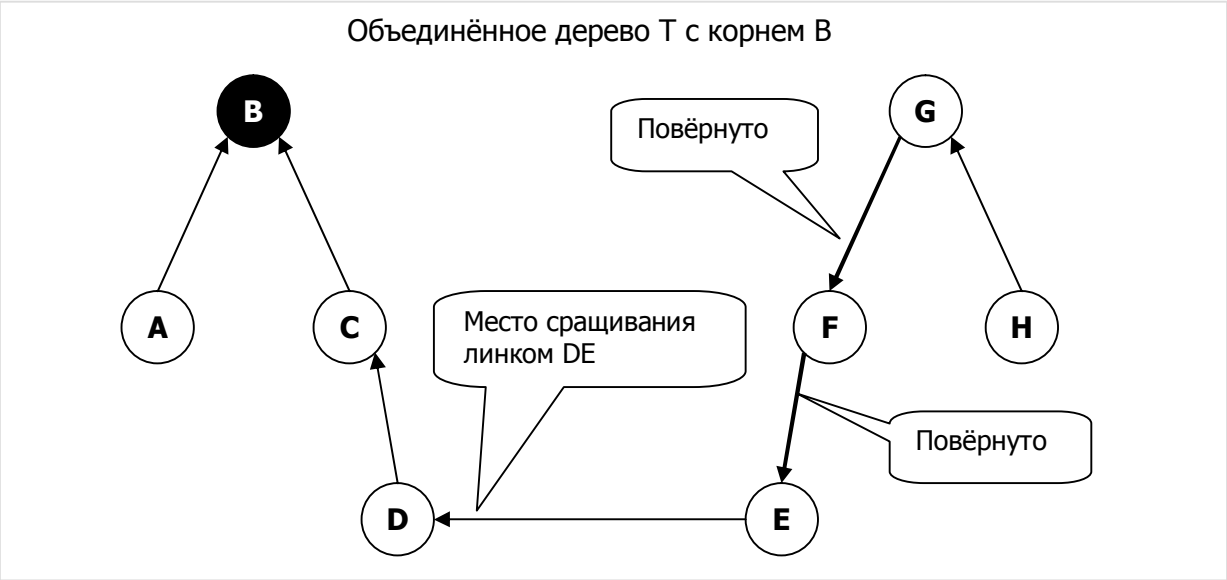


Рис. 23-7 — Состояние после сращивания

Результат сращивания поддеревьев показан на рис. 23-7, процедура сращивания, входящая в состав метода **`ExpoAllTrees`**, представлена в листинге.

Листинг 23-5 — Сращивание двух поддеревьев

```
function Union(aTL: TTreeLink): boolean;
var   N2: TNode;    // узел в T2
      p2: TNode;    // предшествующий узел
      tmp: TNode;   // временное хранилище
begin
  with aTL do begin
    // Если концы линка принадлежат одному поддереву, то выход
    Result:= mOwner.mRoot <> mDest.mRoot;
    if not Result then Exit;
    // Подготовка к реверсу обратных ссылок mPred
    N2:= mDest;         // текущая вершина в поддереве T2
    p2:= N2.mPred;      // предшествующая вершина в T2
    N2.mPred:=mOwner;   // новая предшествующая взята из T1
    // Реверсирование обратных ссылок:
    while Assigned(p2) do begin
      tmp:= p2.mPred;   // временно сохранить ссылку в T2
      p2.mPred:= N2;    // обновить обратную ссылку
      N2:= p2;          // сдвинуть вершину по направлению к 2-му корню
      p2:= tmp;         // восстановить ссылку в T2
    end;
    // Замена 2-го корня на 1-й:
    // root1 = mOwner.mRoot.Expo
    // root2 = mDest.mRoot.Expo
    p2:= mDest.mRoot;
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N2.mRoot = p2 then N2.mRoot:= mOwner.mRoot;
      N2:= NodeNext;
    end;
  end;  // with
end;
```

23.3.5. Расщепление дерева

Теперь займёмся *расщеплением* дерева, оно нужно для восстановления состояния леса при подъёме по дереву перебора. Если последним присоединённым линком был *D-E*, то удаляется именно он (рис. 23-8).

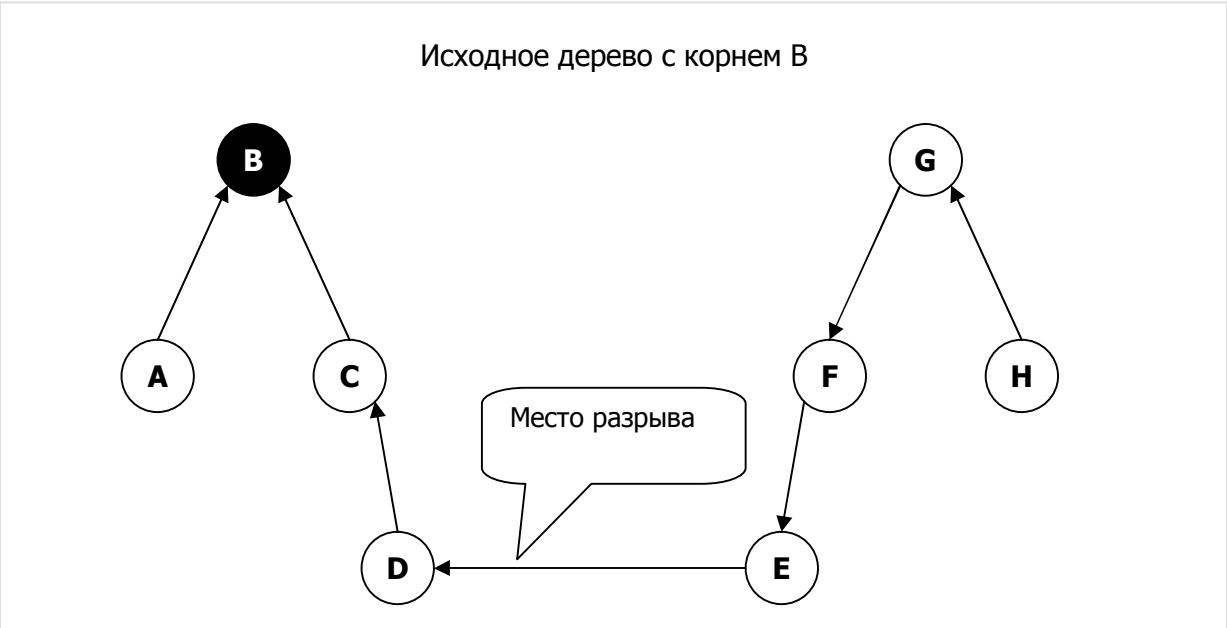


Рис. 23-8 — Состояние перед расщеплением дерева

Здесь, как и при сращивании, никаких операций с левым поддеревом не потребуется, поскольку там «вожак» уже выбран, и все стрелки ведут к нему. Но для правого поддерева надо выбрать корень и направить стрелки в его сторону. Напомню, что нам безразлично, которая из вершин поддерева станет его корнем, и потому выберем ту, с которой будет меньше хлопот. Поскольку все стрелки правого поддерева уже ведут к вершине *E* (а это, заметим, место разрыва), нам выгодно назначить корнем именно эту вершину (рис. 23-9).

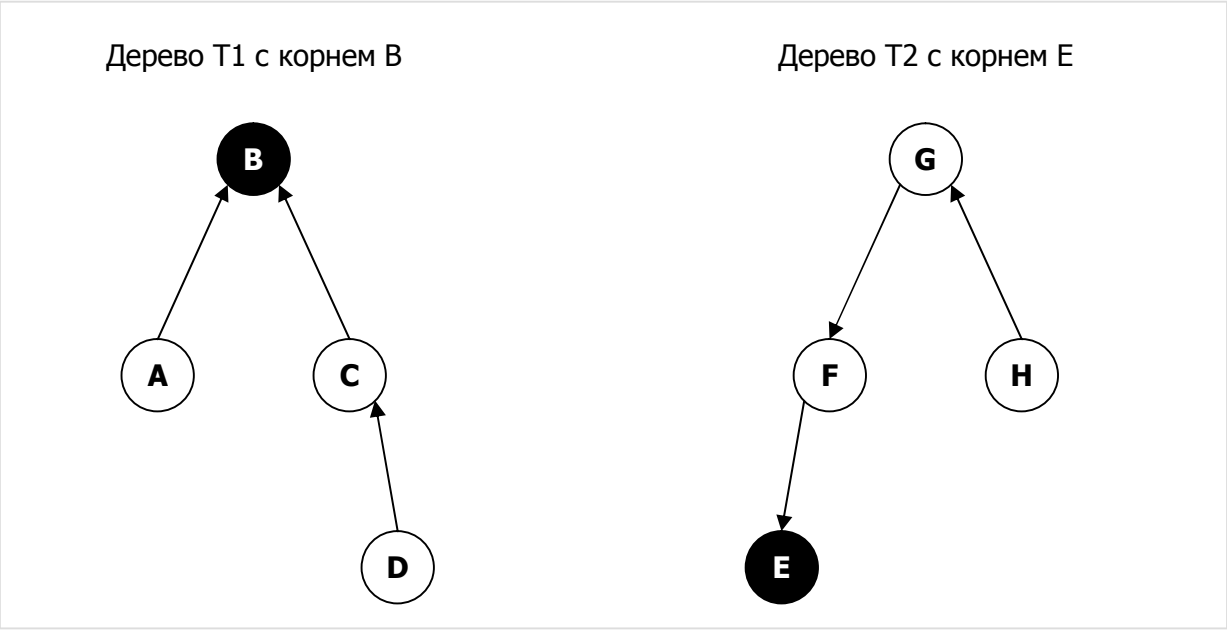


Рис. 23-9 — Состояние после расщепления дерева

Теперь надо пройтись по вершинам правого поддерева *T2* , заменяя прежний корень *B* на новый корень *E*. Здесь, — увы — придётся грести против течения, поскольку стрелки ведут к корню, а не наоборот. Идея состоит в «послойной» обработке вершин. Вначале из всех вершин поддерева *T2* нам известна только корневая вершина: исправим в ней корневую метку **`mRoot`** (будет указывать на неё же) и поместим эту вершину в множество **`S1`**, выполняющее роль *внутреннего слоя*. Затем перебираем все вершины графа в поисках тех, что примыкают своими стрелками (полями **`mPred`**) к этому *внутреннему слою*. Найденные вершины вставляем в множество *внешнего* слоя **`S2`**, попутно заменяя в них корневую метку **`mRoot`**. Если множество **`S2`** окажется пустым, то завершаем процедуру, а иначе *внешний* слой копируем во *внутренний*, внешний слой очищаем и повторяем поиск примыкающих вершин. Листинг 23-6 с процедурой расщепления дерева представлен ниже.

Ещё одно замечание касается поддерева *T2* . Легко заметить, что после расщепления (рис. 23-9) оно не вполне совпадает с тем, что было до сращивания (рис. 23-6), — изменилась и корневая вершина, и направление стрелок. Но эти изменения никак не влияют на решение задачи, поскольку по-прежнему выполняются два требования: все вершины поддерева связаны, и все помечены одним корнем. Да, возможно бывший «вожак стаи» *G* обижен, но тут, как говорится, ничего личного...

Листинг 23-6 — Расщепление дерева

```
  procedure Remove(aTL: TTreeLink);
  var Root: TNode;  // Новый корень для 2-го поддерева
      S1 : TSet;    // внутренний слой вершин
      S2 : TSet;    // внешний слой вершин
      Node: TNode;
  begin
    S1:= CreateSet;    S2:= CreateSet;
    // Выбираем новый корень для 2-го поддерева
    with aTL do if mOwner.mPred = mDest
      then Root:= mOwner    // источник связи
      else Root:= mDest;    // приёмник связи
    Root.mRoot:= Root;      // пометка корня самим собой
    Root.mPred:= nil;       // обратная ссылка в корне пуста
    S1.Insert(Root);        // организуем нулевой слой из корневой вершины
    repeat
      // Накопление внешнего слоя:
      Node:= NodeFirst;
      while Assigned(Node) do begin
        // если примыкает к внутреннему слою, то добавляем к внешнему
        if S1.Exist(Node.mPred) then begin
          Node.mRoot:= Root;
          S2.Insert(Node);
        end;
        Node:= NodeNext;
      end;
      // Выход, если внешний слой пуст
      if S2.GetCount=0 then Break;
      // Копирование внешнего слоя во внутренний и его очистка
      S1.CopyItems(S2);
      S2.Clear;
    until false;
    S1.Free; S2.Free;
  end;
```

23.3.6. Перебор связей

Мы близки к финалу и можем представить листинг 23-7 с методом, перебирающим без повторений все остовные деревья графа. Для экономии места тексты трёх ранее рассмотренных локальных процедур заменены здесь ссылками на листинги.

Листинг 23-7 — Перебор всех остовных деревьев

```
procedure TGraph.ExpoAllTrees;
var
  Links: TSet;        // множество всех линков графа
   Tree: TSet;        // множество линков дерева
  Level: integer;     // текущий уровень
     NN: integer;     // количество линков, обрабатываемых на каждом уровне
    Cnt: integer;     // счётчик остовных деревьев
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация
  procedure Local_Init;
  {Листинг 23-4 }
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Объединение двух поддеревьев
  function Union(aTL: TTreeLink): boolean;
```

```
  {Листинг 23-5}
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление ребра и расщепление дерева
  procedure Remove(aTL: TTreeLink);
  {Листинг 23-6}
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вспомогательная функция вычисления стоимости остова
  // как суммы весов его рёбер
  function Cost(aTree: TSet): integer;
  var TL: TTreeLink;  // текущий линк для дерева
  begin
    Result:= 0;
    TL:= aTree.GetFirst as TTreeLink;
    while Assigned(TL) do begin
      Inc(Result, TL.mValue);
      TL:= aTree.GetNext as TTreeLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный перебор всех подмножеств (комбинаций) из L по N-1 линков
  // L - количество рёбер или дуг графа = Links.GetCount
  // N - количество вершин графа = mNodes.GetCount
  procedure Searching(aIndex: integer { стартовый индекс } );
  var  i: integer;
       TL: TTreeLink;  // текущий линк для дерева
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      TL:= Links.GetItem(i + Level) as TTreeLink;  // очередной линк
      if Union(TL) then begin
        // Этот линк не создаёт цикла, вставляем в остов
        Tree.Insert(TL);
        if Level = mNodes.GetCount-2 then begin
          // Остов сформирован, вывод:
          Inc(Cnt);       // счётчик остовных деревьев
          Write(Cnt:4, '':2, Cost(Tree):3, '':2);
          Tree.Expo;
        end else begin
          // Здесь размер подмножества не достигнут, остов не сформирован,
          // погружаемся на следующий уровень
          Inc(Level);    // Level+1
          Searching(i);  // рекурсивный вызов следующего уровня
          Dec(Level);    // Level-1
        end;
        // Возвращаем предыдущее состояние поддеревьев
        Tree.Delete(TL);  // удаляем из остовного множества
        Remove(TL);       // и расщепляем дерево
      end; // if
    end; // for
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin
  Local_Init;
  Cnt:= 0;     // счётчик остовных деревьев
  Level:=0;    // текущий уровень
  // количество линков, обрабатываемых на каждом уровне:
  // (mNodes.GetCount - 1) = количество рёбер дерева
  NN:= Links.GetCount + 1 - (mNodes.GetCount - 1);
  Searching(1);
  Links.ClrAndDestroy;
  Links.Free;
end;
```

23.4. Испытание

Испытание методов построения минимального остова и перебора всех остовов выполнено на следующем графе:

```
Test.txt
0 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

А это тестирующая программа:

Листинг 23-8 — Программа для тестирования методов поиска остовных деревьев

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Tree: TSet;
    Cost: integer;
begin
  Gr:= TGraphChars.Load('Test.txt');
  Gr.Expo;
  Writeln('------------------------------');
  Tree:= Gr.GenCoverTree(false, Cost); // минимальный остов
  Tree.Expo;
  Writeln('Cost= ', Cost);
  Tree.ClrAndDestroy;  Tree.Free;
  Writeln('------------------------------');
  Tree:= Gr.GenCoverTree(true, Cost); // максимальный остов
  Tree.Expo;
  Writeln('Cost= ', Cost);
  Tree.ClrAndDestroy;  Tree.Free;
  Gr.ExpoAllTrees;           // все остовные деревья
  Readln;
end.
```

Ниже показана первая и последняя десятка из 118 остовных деревьев, полученных в ходе перебора:

```
   1   36  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 BC= 8 } : 7
   2   36  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AF= 8 } : 7
   3   38  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AG= 10 } : 7
   4   38  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BG= 7 AE= 10 } : 7
   5   39  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 BC= 8 AG= 10 } : 7
   6   39  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 AF= 8 AG= 10 } : 7
   7   41  { AC= 1 EF= 4 BH= 4 AD= 6 EH= 6 AG= 10 AE= 10 } : 7
   8   38  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 BC= 8 AF= 8 } : 7
   9   40  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 BC= 8 AE= 10 } : 7
  10   40  { AC= 1 EF= 4 BH= 4 AD= 6 BG= 7 AF= 8 AG= 10 } : 7
. . .
 109   53  { BH= 4 AD= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 110   49  { BH= 4 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 } : 7
 111   49  { BH= 4 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AE= 10 } : 7
 112   52  { BH= 4 CD= 6 EH= 6 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 113   53  { BH= 4 CD= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 114   51  { AD= 6 CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AE= 10 } : 7
 115   53  { AD= 6 CD= 6 EH= 6 BG= 7 AF= 8 AG= 10 AE= 10 } : 7
 116   54  { AD= 6 CD= 6 EH= 6 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 117   55  { AD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
 118   55  { CD= 6 EH= 6 BG= 7 BC= 8 AF= 8 AG= 10 AE= 10 } : 7
```

Деревья следуют в порядке возрастания их стоимости (веса), что объясняется порядком сортировки вспомогательных линков. При соответствующем изменении метода **`TTreeLink.Compare`** этот порядок можно обратить, а также построить метод поиска *максимального* остова, взяв за основу метод **`GenMinCover`**, и изменив там некоторые условия.

К этому добавим, что количество всех мыслимых комбинаций из **`11`** рёбер по **`7`** составляет **`330`**, но только **`118`** из них дали то, что мы ищем — графы, не содержащие циклов.

23.5. Ориентированные деревья

Итак, мы построили остовные деревья для *неориентированных* связанных графов (в несвязанном графе остов не построить). А как обстоит дело с орграфами? Здесь нам интересен ориентированный остов, то есть, ориентированное дерево с выделенной вершиной — корнем. Поэтому для начала ознакомимся с особенностями и свойствами таких деревьев.

Определения, данные прежде для неориентированного дерева, справедливы и здесь, но в силу ориентации связей взаимная достижимость всех вершин тут невозможна. Зато особыми правами наделена вершина, называемая корнем (**`Root`**). В зависимости от направления, связи могут вести либо от корня к периферийным вершинам, либо, наоборот — от периферии к корню (рис. 23-10). Назовём эти деревья *прямым* и *инверсным*. В *прямом* дереве корень недостижим ни из одной вершины, но все вершины достижимы из корня. В *инверсном*, наоборот: корень достижим из любой вершины, но, ни одна периферийная вершина не достижима из него. В *прямом* дереве полустепень *захода* любой вершины, кроме корня, равна единице (есть только одна входящая связь). В *инверсном* дереве это справедливо для полустепени *исхода*.

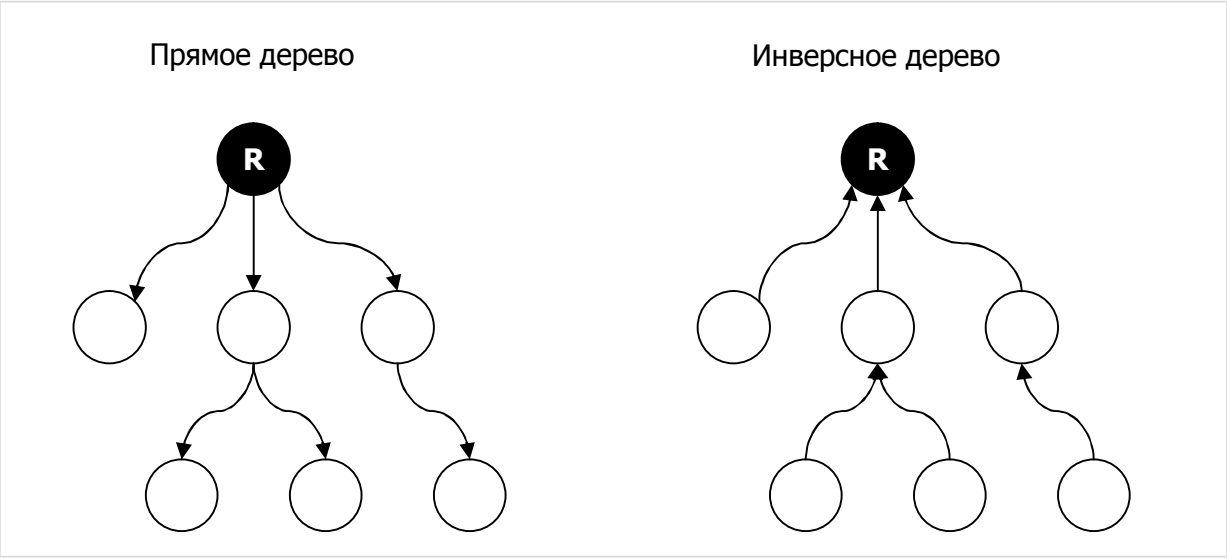


Рис. 23-10 — Ориентированные деревья с выделенным корнем

Теперь поведаю историю, когда постройка минимального ориентированного остова решила судьбу государства. Обратимся для этого к рис. 23-11, где показан архипелаг и мосты с односторонним движением.

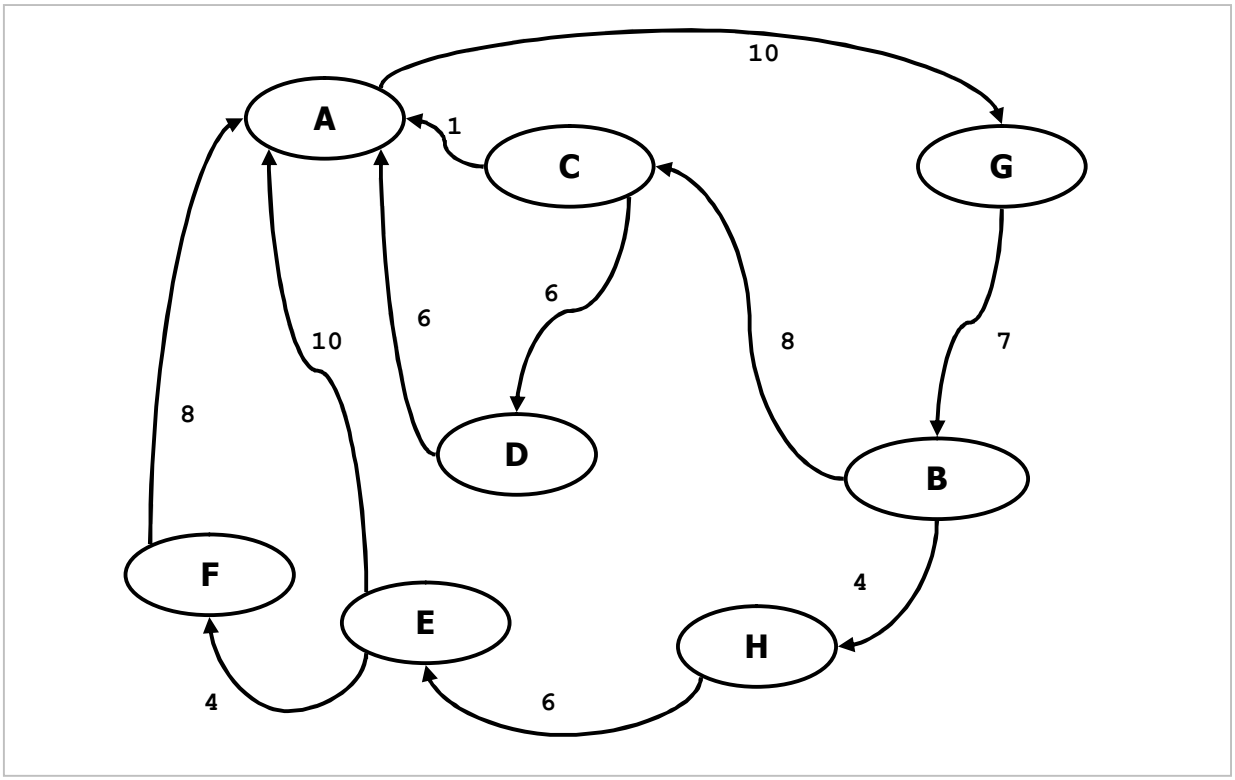


Рис. 23-11 — Ориентированный граф

Здесь угадывается сильно связанный орграф, то есть такой, где все вершины взаимно достижимы. Однажды правительство — в целях экономии — решило вывести из эксплуатации часть мостов, но так, чтобы граф остался связным, хоть и не сильно связным. Как тогда быть с взаимной достижимостью островов,

необходимой для нормальной жизни? Решили прибегнуть к периодической *инверсии*: направление движения на оставшихся мостах время от времени менять на противоположное.

Эта реформа заставила выбрать новый столичный остров с тем условием, чтобы гарантировать почтовую связь столицы со всеми островами, а эта связь была организована так. Посыльные, выехав из столицы, доставляли к *ближайшим* соседям служебную почту, а позже (после инверсии) возвращались с донесениями назад. Далее, на периферийные острова почта попадала точно также, но с другими посыльными, курсирующими между соседними островами. Таким образом, на каждый остров со стороны центра прибывал ровно *один* посыльный, а на периферию из вершины их отправлялось столько, сколько было у данного острова ближайших пери соседей. Ваша догадка верна: система являла собой дерево с выделенной вершиной — столицей архипелага. И задача состояла в том, чтобы выбрать столицу (корень) и множество мостов (ориентированный остов) с минимальной общей длиной.

Первый вопрос таков: будем ли мы искать *прямое* дерево, или *инверсное* ? Поскольку ориентация их дуг взаимно противоположна, и одно легко получается из другого, сосредоточимся на поиске *прямого* дерева. Легко сообразить, что выбрав в качестве корня одну из вершин графа, все последующие можно «подцепить» к ней кратчайшими *исходящими* дугами, следуя алгоритму Прима. Поскольку орграф с выделенной вершиной не сильно связан, надо быть готовым к тому, что далеко не каждый корень породит покрывающий остов. Мало того, может случиться и так, что в ходе перебора корней ни один покрывающий ориентированный остов так и не сформируется (рис. 23-12).

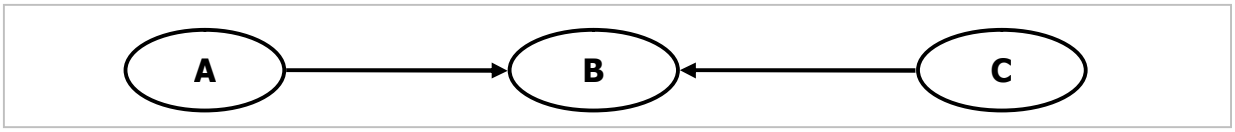


Рис. 23-12 — Для этого графа не существует направленного остова с выделенной вершиной

Ниже представлен метод поиска минимального покрывающего ориентированного остова и соответствующего ему корня.

Листинг 23-9 — Поиск минимального ориентированного остова с выделенной вершиной (алгоритм Прима)

```
function TGraph.GenCoverDir(var aRoot: TNode;    // корень
                            var aCost: integer   // вес (стоимость) дерева
                           ): TSet;              // само дерево
var Tree: TSet;      // текущее дерево
    Links: TSet;     // текущее множество линков
  //.....................................................
  // Локальная функция формирует дерево, начиная с вершины aRoot
  // алгоритмом Прима и возвращает его вес (стоимость).
  // Не сформированное дерево очищается
  function RootCover(aRoot: TNode): integer;
  var Node: TNode;       // очередная вершина
```

```
      TL : TTreeLink;     // очередной линк
  begin
    Result:= 0;           // вес дерева
    ResetNodes;           // для всех вершин mColor = CWhite
    Node:= aRoot;         // исходная вершина -- корень
    // Пытаемся присоединенить все вершины графа (линков на единицу меньше)
    while Assigned(Node) and
         (Tree.GetCount < mNodes.GetCount-1) do  begin
      Node.mColor:= CBlack;      // текущую в множество чёрных
      // добавляем линки к ближайшим белым вершинам:
      Node.AddTreeLinks(false, Links);
      // Перебираем линки Links в поиске ближайшей белой вершины
      TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
      Node:= nil;                         // искомая вершина пока пуста
      while Assigned(TL) do begin         // пока существуют линки
        Links.Delete(TL);                 // удаляем линк из буфера Links
        // Искомая вершина может быть только приёмником связи:
        if TL.mDest.mColor = CWhite then begin
          // Здесь ближайшая белая вершина найдена:
          Node:= TL.mDest;        // будет обработана в следующем цикле
          Tree.Insert(TL);        // вставляем линк в результат
          Inc(Result, TL.mValue); // наращиваем стоимость
          Break;
        end;
        // здесь линк связывает две чёрные вершины:
        TL.Free;                          // удаляем ненужный линк
        TL:= Links.GetNext as TTreeLink; // и берём следующий
      end;
    end;
    Links.ClrAndDestroy; // очистка оставшихся в очереди линков
    // Если остов не построен, очищаем накопитель линков
    if Tree.GetCount < mNodes.GetCount-1 then begin
      Tree.ClrAndDestroy;  // очистка поддерева
      Result:= MaxInt;
    end;
  end;
  //.....................................................
var Root: TNode;    // очередной корень дерева
    Cost: integer;  // стоимость (вес) дерева
Begin  { TGraph.GenCoverDir }
  Result:=nil;  aRoot:= nil;  aCost:= MaxInt;
  Links:= CreateSet;  // текущее множество линков
  Tree:= CreateSet;   // текущее остовное дерево
  // Перебор всех вершин:
  Root:= NodeFirst;
  while Assigned(Root) do begin
    Cost:= RootCover(Root);
    // Если дерево построено:
    if Tree.GetCount > 0 then begin
      // Если стоимость меньше максимальной
      if Cost<aCost then begin
        // то сохраняем результат
        if Assigned(Result)
          then Result.ClrAndDestroy // удаление существующих линков
          else Result:=CreateSet;   // создание копии дерева
        Result.CopyItems(Tree);     // копируем дерево (линки)
        Tree.Clear;                 // удаляем сохранённые линки
        aRoot:= Root;               // запоминаем корень
        aCost:= Cost;               // запоминаем стоимость (вес)
      end else begin
        Tree.ClrAndDestroy;  // очистка дерева, если оно не минимально
      end;
```

```
    end;
    Root:= NodeNext;
  end;
  // Удаление рабочих буферов:
  Tree.Free;
  Links.Free;
end;
```

Метод тестировался следующей программой:

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;       // исходный граф
    Tree: TSet;        // множество дуг дерева
    Root: TNode;       // корень дерева
    Cost: integer;     // стоимость дерева

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Tree:= Gr.GenCoverDir(Root, Cost);
    if Assigned(Tree) then begin
       Root.Expo;
       Tree.Expo;
       Writeln('Cost= ', Cost);
       Tree.ClrAndDestroy;
       Tree.Free;
    end;
    Readln;
end.
```

Для испытаний взят граф, изображённый на рис. 23-11:

```
Test.txt
1 - тип графа (1 = орграф)
1 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

В результате получен минимальный остов с корнем в G и стоимостью 36, показанный на рис. 23-13. Пунктир обозначает выведенные из эксплуатации мосты.

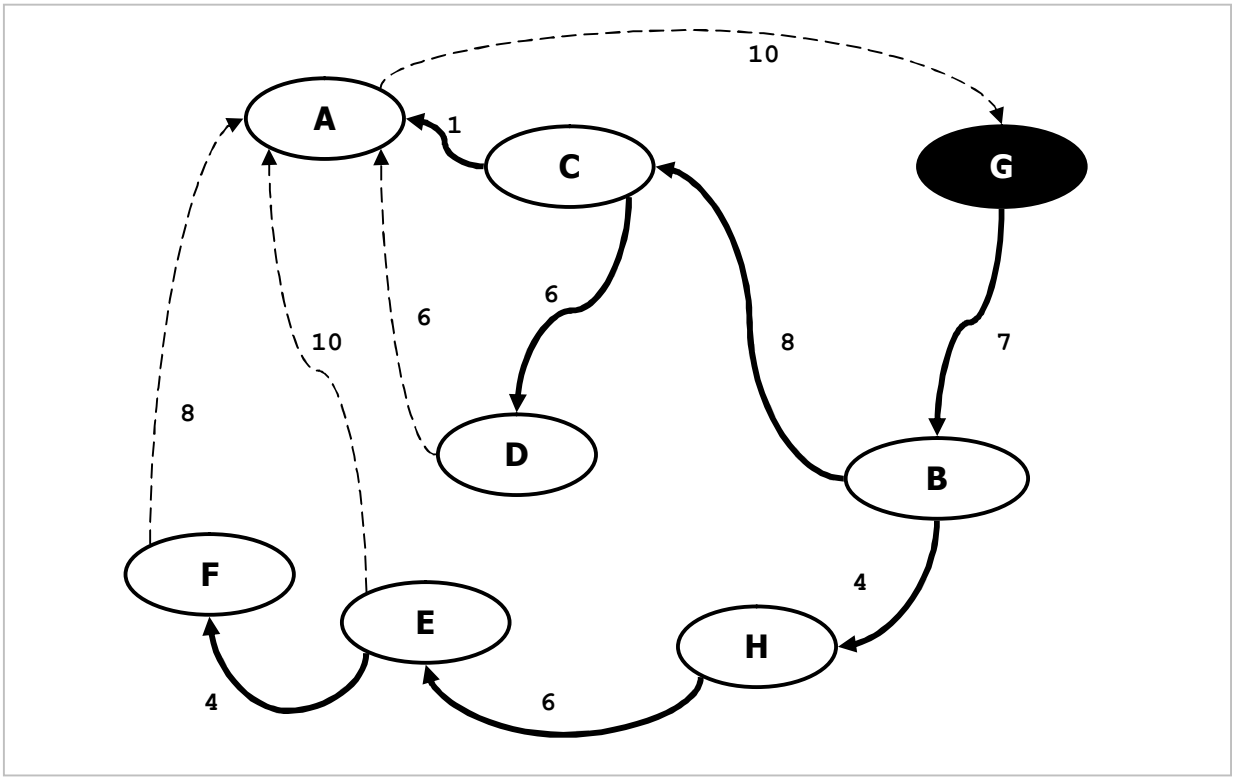


Рис. 23-13 — Минимальный ориентированный остов с корнем в G и стоимостью 36

23.6. Итоги

23.6.1. Остовным деревом графа (остовом) называется связанный граф, содержащий N-1 ребро (N— число вершин графа). Остове не содержит циклов.

23.6.2. Каждое ребро или дуга остовного дерева соответствует ребру или дуге исходного графа или орграфа.

23.6.3. Количество остовов в графе может быть весьма велико, и в общем случае задача перебора всех остовов экспоненциально сложна.

23.6.4. Алгоритм Прима находит минимальный либо максимальный остов графа, — в зависимости от порядка сортировки вспомогательных линков и условий проверки.

23.6.5. Поиск всех остовов графа состоит в переборе комбинаций по N-1 линков, при этом линки проверяются на возможность образования циклов. В ходе перебора попеременно происходит соединение и расщепление поддеревьев.

23.6.6. В ориентированном графе при соответствующих условиях могут быть построены ориентированные остовы с выделенной вершиной — корнем.

23.7. Задания

23.7.1. Формирование остовов неориентированного графа

Сконструируйте метод, формирующий, в зависимости от параметров, либо минимальный, либо максимальный остов.

23.7.2. Формирование остовов орграфа

Сделайте то же самое для формирования минимального либо максимального ориентированного остова с выделенной вершиной.

23.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 162
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 24
Максимальный поток

В этой и двух последующих главах рассмотрим задачи о *потоках* в сетях. *Сетью* называют ориентированный граф, каждой дуге которого приписана некоторая пропускная способность. Так моделируют перенос всякого рода материальных или информационных объектов: автомобилей по улицам, жидкостей по трубам, пассажиров по маршрутам. Пропускная способность дуг выражается единицами объектов в единицу времени: автомобилями в час, кубометрами в минуту, пакетами в секунду и т.п.

Вот город на оживлённой трассе. Его перекрёстки уподобим вершинам, а улицы — дугам орграфа. Двустороннее движение по улицам моделируем парами дуг, причём пропускная способность дуги соотносится с количеством полос (встречные дуги в принципе могут обладать разной пропускной способностью). Пусть транспортный поток (одного направления) вливается в город с трассы через вершину S, растекается по улицам, а затем выходит на трассу через вершину T. Здесь интересно знать: 1) какова максимально возможная величина этого потока, и 2) как распределить поток по улицам с тем, чтобы достичь этого максимума? Так формулируются основные задачи о *максимальном потоке*, алгоритмы их решения лежат в основе решений других разновидностей потоковых задач.

24.1. Основные понятия

На рис. 24-1 показан орграф, отражающий суть обсуждаемой транспортной задачи. В нём выделены две вершины: исток S и сток T. Все прочие вершины — промежуточные. Пропускная способность указана рядом с дугами и в некоторой степени отражена толщиной дуг.

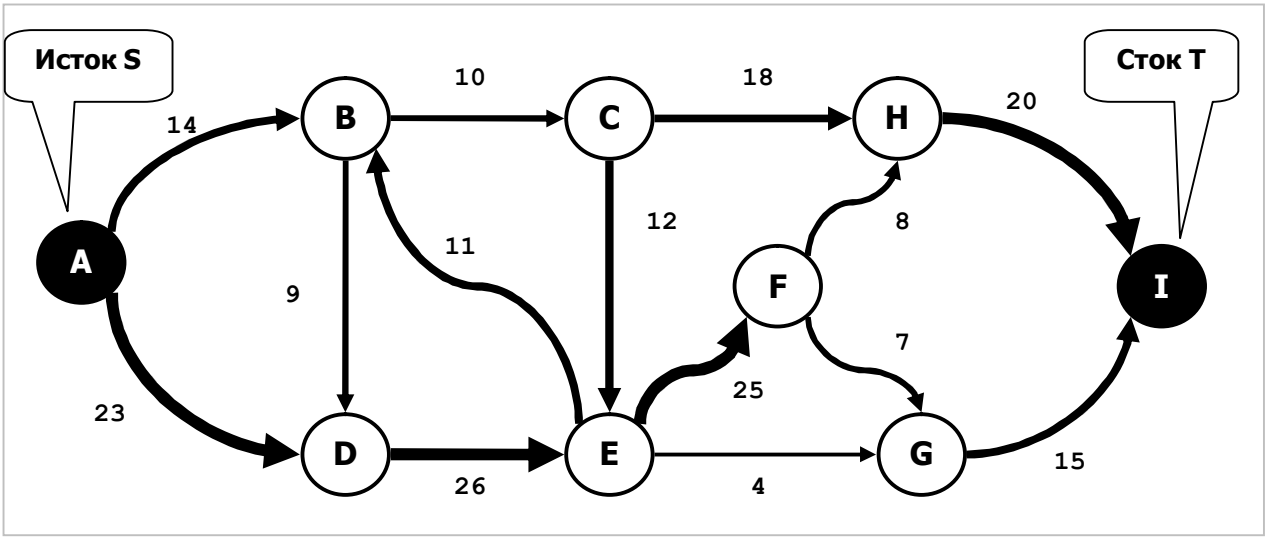


Рис. 24-1 — Граф для транспортной задачи с источником S и стоком T

Вершина-исток S генерирует некие объекты и отправляет их в сеть. Соответственно вершина-сток T поглощает эти объекты. Все прочие вершины (промежуточные) только пропускают поток в направлении дуг. Отсюда следуют очевидные утверждения:

- Количество транспортируемых элементов, входящих в промежуточную вершину, в точности равно количеству исходящих из неё. То есть, сумма потоков для промежуточной вершины равна нулю, если входящим и исходящим потокам приписаны разные знаки.
- Максимальный поток, передаваемый из источника в сток, не может превышать сумму пропускных способностей дуг, инцидентных *истоку*.
- Этот же максимальный поток не может превышать сумму пропускных способностей дуг, инцидентных *стоку*.
- Максимальный поток, следующий вдоль некоторой цепочки дуг от источника к стоку, ограничен пропускной способностью самой «узкой» дуги в этой цепочке (*критической* дуги).

24.2. Способы увеличения потока

Идея решения задачи о максимальном потоке состоит в постепенном наращивании потока, начиная с некоторого минимума. В этой связи рассмотрим два пути увеличения потока, протекающего от источника к стоку. На рис. 24-2 представлен упрощённый граф, где стоку T отведена роль пресловутого бассейна с двумя трубами, через одну из которых — прямую — вода поступает в бассейн, а через другую — обратную — вытекает и возвращается в исток.

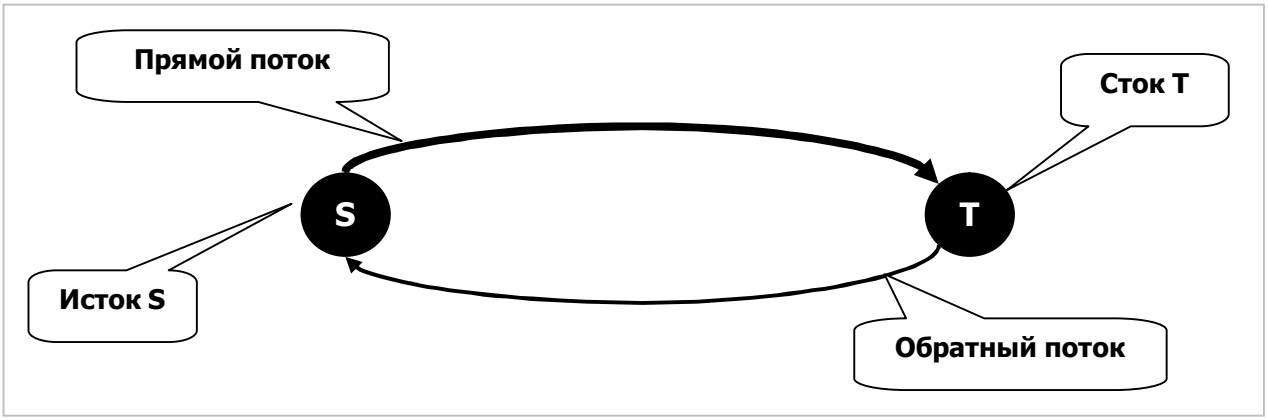


Рис. 24-2 — Упрощённая схема потоковой задачи

Пусть на некотором этапе решения задачи вода течёт в обоих направлениях, как здесь ускорить наполнение бассейна T? Очевидно, что есть два пути:

- Во-первых, если позволяет прямая труба, увеличить прямой поток.
- Во-вторых, уменьшить (вплоть до нуля) поток в обратной трубе.

Разумеется, что в реальном графе картина течений сложнее, но основные идеи те же: мы будем искать такие цепочки дуг, вдоль которых можно либо *увеличить прямой* поток, либо *уменьшить обратный*.

24.3. Увеличивающие, уменьшающие и нейтральные дуги

Вспомним, что *цепью* называют ряд попарно инцидентных дуг, соединяющий исходную и конечную вершины (*исток* и *сток*). Причём дуги цепи не обязательно направлены от истока к стоку. Перед составлением нужных нам *увеличивающих* цепочек дуг, распределим все дуги на три класса (множества). Это распределение будет меняться по ходу решения задачи, поскольку отнесение дуги к тому или иному классу зависит и от протекающего в ней потока, и от направления дуги в выбранной цепи. Заметим, что одна и та же дуга в одной цепочке может оказаться прямой, а в другой цепочке — обратной.

Прямые дуги, поток через которые ниже предела пропускной способности дуги (не насыщен), отнесём к классу *I* (Increase). Через такую дугу можно увеличить поток от истока к стоку.

К классу *R* (Reverse) — отнесём *обратные* дуги с *ненулевым* потоком. Уменьшив утечку через такую обратную дугу, можно в целом увеличить потребление в стоке.

Наконец к третьему, *нейтральному* классу *N*, отнесём прочие дуги, то есть: 1) *прямые* дуги с *насыщенным* потоком, и 2) *обратные* с *нулевым* потоком. Любое изменение потока в них неминуемо уменьшит поток от источника к стоку, — нейтральные дуги *не будем* включать в искомые цепочки.

Осталось выяснить, как отличить *прямые* дуги от *обратных* ? Алгоритмы, которые мы скоро построим, будут перебирать вершины графа так, что по отношению к некоторой промежуточной вершине *X* все *исходящие* из неё дуги окажутся *прямыми*, а все *входящие* — *обратными* (см. рис. 24-3).

На рис. 24-3 показана промежуточная вершина, у которой для составления увеличивающей цепи интерес представляют две верхние дуги: *входящая* с *ненулевым* потоком, и *исходящая* с *ненасыщенным* потоком. В последней резерв потока составляет **7−3=4** единицы. Две нижние дуги — нейтральные. В левой входящей нельзя уменьшить поток, поскольку его там нет, а в правой исходящей поток нельзя увеличить, поскольку он насыщен (резерв составляет **5−5=0** единиц).

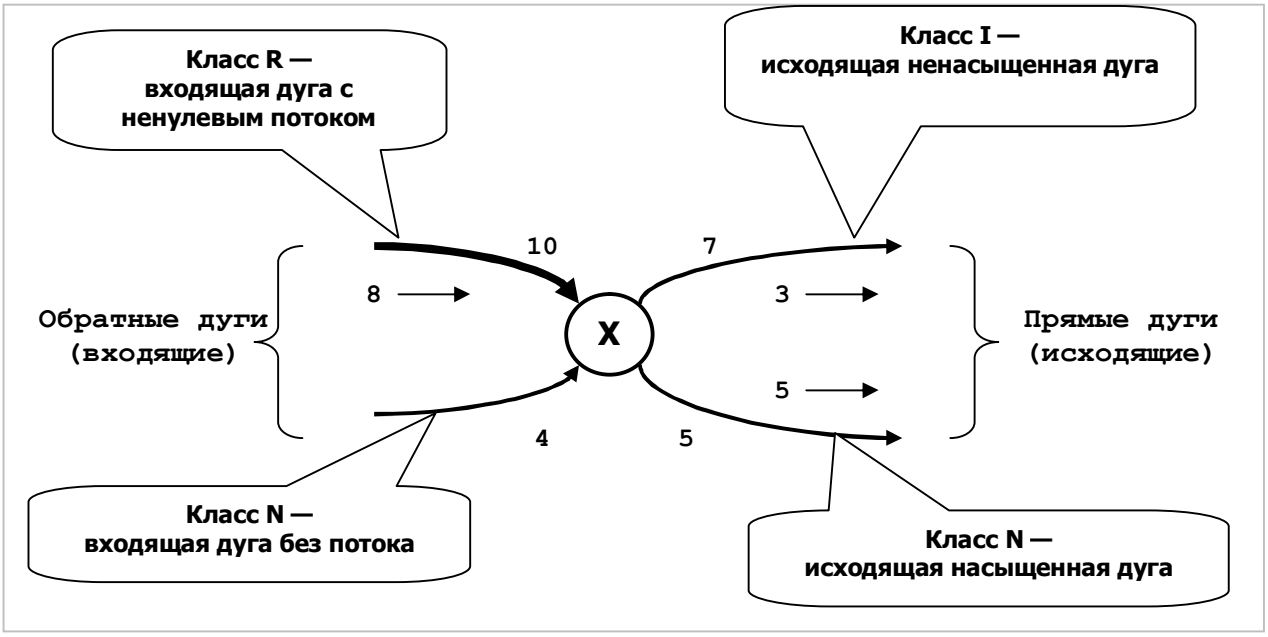


Рис. 24-3 — Классы дуг по отношению к вершине X

Повторю, что поскольку порядок обработки вершин в ходе решения задачи будет меняться, одна и та же дуга в разных цепочках может оказаться как прямой, так и обратной.

24.4. Увеличивающие цепочки

Составлением увеличивающих цепей займёмся позже, а пока рассмотрим сами эти цепочки. Пусть нам удалось найти увеличивающую цепь, составленную только из *прямых* дуг класса *I* (рис. 24-4). Поток, текущий по дугам, показан стрелками. Ясно, что вдоль этой цепочки поток можно увеличить лишь на **3** единицы, поскольку таким минимальным запасом обладает дуга *Y-Z* (**5−2=3**).

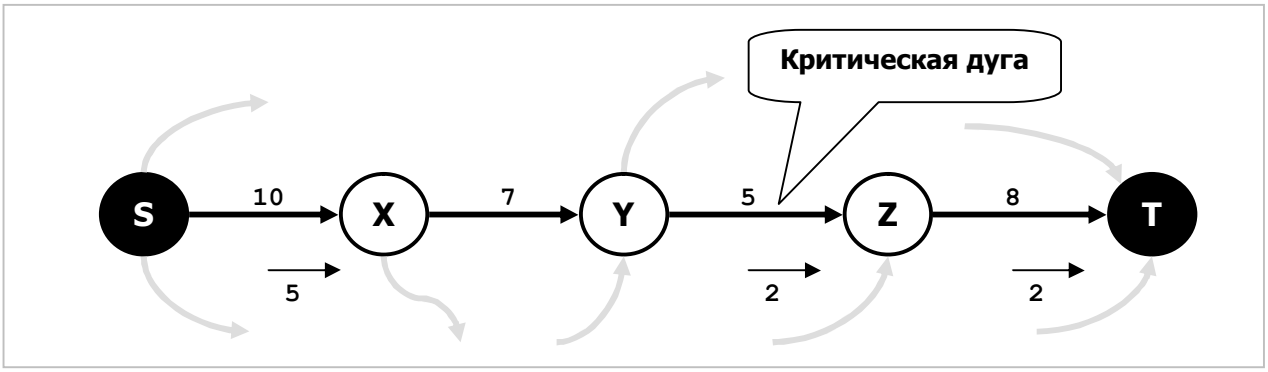


Рис. 24-4 — Увеличивающая цепь, составленная из прямых дуг класса I

Теперь рассмотрим цепь на рис. 24-5, она составлена только из *обратных* дуг, по которым «стекает назад» ненулевой поток. Снизить этот отток можно не более чем на **2** единицы, и тем самым на **2** единицы увеличится потребление объектов в стоке *T*.

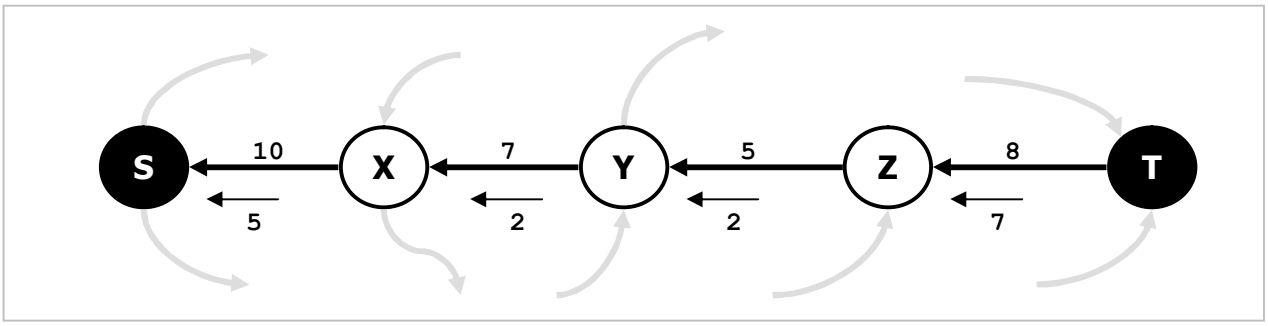


Рис. 24-5 — Увеличивающая цепь, составленная из обратных дуг класса R

Впрочем, цепочек, составленных только из обратных дуг, наш алгоритм не обнаружит. Зато возможны *смешанные* цепочки, содержащие как *прямые*, так и *обратные* дуги, вроде той, что показана на рис. 24-6.

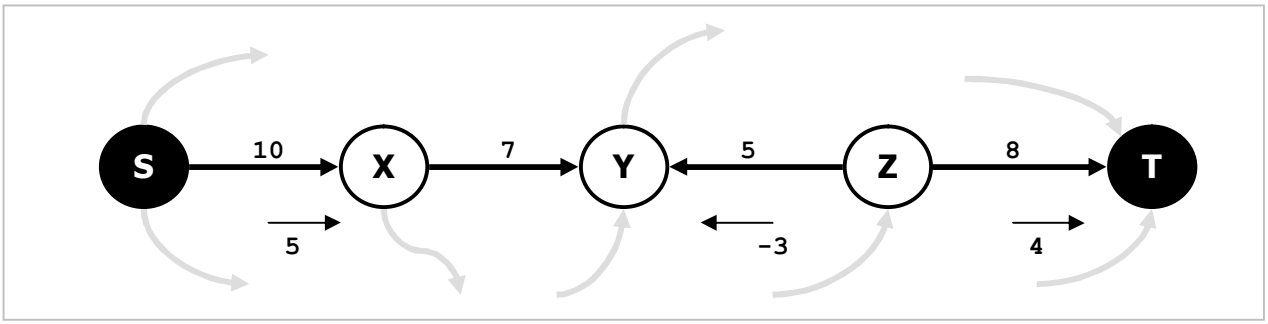


Рис. 24-6 — Увеличивающая цепь, составленная из прямых и обратных дуг

Здесь обратим внимание на дугу *Z-Y*, по которой в обратном направлении движется поток в минус **3** единицы. Эти единицы потока не попадают напрямую в источник *S*, но создают местную циркуляцию, нагружая ею прямые дуги и затрудняя тем самым прямое движение потока. До какой степени можно увеличить поток вдоль такой цепи? Для ответа пройдём вдоль цепи и найдём минимальное приращение потока, которое:

а) не приведёт к превышению пропускной способности *прямых* дуг;
б) и не превзойдет поток в *обратных* дугах (тут потоку в обратной дуге условно приписан знак минус).

Легко догадаться, что здесь поток можно увеличить на **3** единицы, после чего распределение потока вдоль цепи станет таким, как показано на рис. 24-7. Теперь поток в обратной дуге *Z-Y* иссяк, зато в прямых дугах он возрос. В конечном счете, общий поток от источника к стоку тоже возрос на **3** единицы. Учтите, что рассматриваемый тут поток обеспечен течениями и в других дугах сети, они отмечены на рисунке серым цветом.

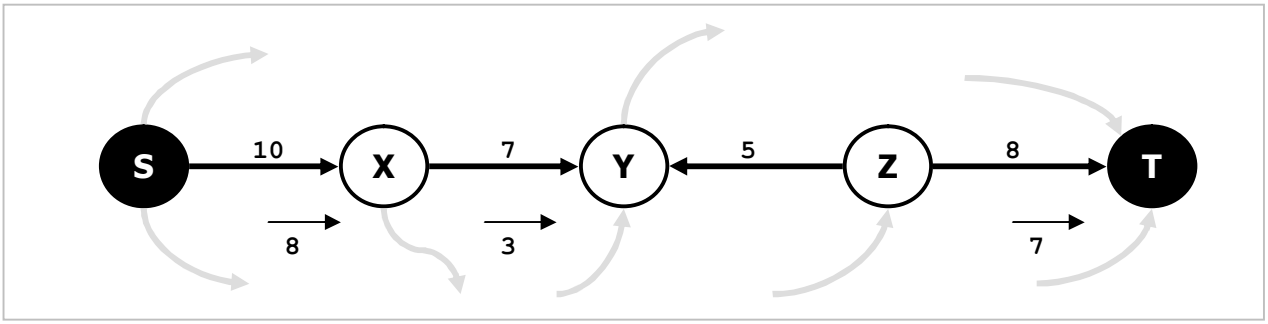


Рис. 24-7 — Поток вдоль цепи после его увеличения на 3 единицы

Отсюда виден принцип решения задачи, и состоит он в следующем. Первоначально вдоль всех дуг графа пускаем нулевой поток. Далее отыскиваем увеличивающую цепь из дуг класса *I* и *R*, и наращиваем вдоль неё поток на предельно возможную для этой цепи величину. Если такую цепь найти не удаётся, алгоритм завершается, а иначе повторно ищем увеличивающую цепь с учётом уже текущего в дугах потока. Теперь обратимся к поиску увеличивающих цепей.

24.5. Поиск увеличивающей цепи

Фантазия найдёт много причудливых цепочек между источником и стоком сети. Но интуиция предпочитает простоту: будем искать *кратчайшие* цепи, ведущие от источника к стоку. То есть цепи, количество дуг в которых минимально. Подобная задача уже решалась нами в главе 15 при поиске кратчайших путей между вершинами (когда длина рёбер не учитывается).

Здесь поступим аналогично: в ходе поиска очередной цепи будем, начиная с истока *S*, расширять гамму вершин до тех пор, пока в этой гамме окажется вершина-сток *T* (вершины гаммы будем окрашивать). Попутно сформируем обратные ссылки и определим минимально допустимое увеличение потока вдоль цепи. Затем по обратным ссылкам на предыдущую вершину пройдём вдоль цепочки в обратном направлении, увеличивая поток вдоль прямых дуг, и уменьшая его вдоль обратных. Так действует одна итерация алгоритма, по завершении которой в графе появится, по меньшей мере, либо одна *насыщенная прямая* дуга, либо одна *истощённая обратная* дуга. Эти нейтральные дуги в следующем цикле игнорируются, и потому следующая итерация породит уже другую кратчайшую цепь, либо не породит её вовсе. В последнем случае алгоритм завершается.

24.6. Данные о потоке

Теперь, когда алгоритм в целом прояснился, дополним нужными полями вершины и дуги графа. Для вершин такими полями будут **`mFlow`** — поток, протекающий через вершину, и **`mLink`** — обратная ссылка в цепочке.

Пять новых полей в дугах, вводимые для решения потоковых задач, представлены ниже. Здесь два из них — **`mLow`** и **`mTemp`** — будут использованы в следующих потоковых задачах, где и будут обсуждены подробней.

```
  // Линк -- ребро или дуга
  TLink = class (TItem)
      ...
      // Для решения потоковых задач:
      mLow   : integer; // допустимые минимальный поток  mLow <= mHigh
      mHigh  : integer; // допустимые максимальный поток
      mFlow  : integer; // текущий поток
      mTemp  : integer; // для временного хранения mHigh
      mDirect: integer; // направление дуги (+1 -- прямая, -1 -- обратная)
      ...
  end;
```

24.7. Ввод данных о потоке

Следующий шаг на пути создания потоковых алгоритмов — построение метода для ввода данных о потоке из текстового файла. Сейчас нас интересует, в первую очередь, максимальная пропускная способность дуг, а в последующем к ней присоединятся минимальная пропускная способность и стоимость дуги. К настоящему моменту нами отработан механизм ввода графов разного типа, но данные о потоках там не учтены. Чтобы не ломать уже готовые и отработанные процедуры, для ввода потоковых данных создадим ещё один метод:

`TGraphChars.LoadFlowData(const aName: String)`

Он читает данные о потоке из указанного текстового файла, причём в качестве такового может служить и файл, содержащий основные данные о графе. Начало потоковых данных пометим волшебной меткой «FLOW», за которой последуют строки с данными, включающими дугу, а также её минимальную и максимальную пропускные способности, например:

```
FLOW:
AB= 1 2
AC= 1 3
BC= 0 1
BD= 1 4
```

Здесь в первой после метки строке даны параметры потока для дуги *A-B*, в следующей строке — для дуги *A-C*, и так далее. За подробностями реализации процедуры отсылаю к модулю **`GrChars`**.

24.8. Функция поиска максимального потока

Итак, разобравшись с хранением и загрузкой потоковых данных, приступим к главному — процедуре поиска максимального потока для случая, когда минимальный поток не задан, то есть, равен нулю. Более сложный случай — с ненулевыми минимальными потоками — рассмотрим в следующей главе. Поскольку алгоритм разобран нами достаточно подробно, приведём текст соответствующего метода, снабдив его лишь несколькими дополнительными пояснениями.

Листинг 24-1 — Поиск максимального потока при отсутствии нижней границы

```
function TGraph.CalcMaxFlow0(aSource, aDest: TNode): integer;
var Delta: integer; // очередное приращение потока
    Que: TBuffer;   // очередь вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт начального потока
  // на случай, когда заданы минимальные потоки в сети (mLow>0)
  function CalcStartFlow: integer;
  var Link: TLink;    // исходящая дуга
  begin
    Result:= 0;
    Link:= aSource.OutLinkFirst;
    while Assigned(Link) do begin
      Inc(Result, Link.mFlow);
      Link:= aSource.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Построение увеличивающей цепи
  // с подсчётом соответствующего приращения потока
  function CalcDelta: integer;
  var Node: TNode;
      Link: TLink;    // исходящая или входящая дуга
  begin
    Result:=0;
    // установка mColor=0, mPred=nil, mDist= MaxInt, mFlow= MaxInt
    ResetNodes;
    Que.Clear;   // очистка очереди вершин
    // Пометку вершин начинаем с источника
    aSource.mColor:= CGray;
    Que.Put(aSource);
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Выход из цикла, если помечен сток
      if Node=aDest then Break;
      // Обработка исходящих связей текущего узла:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины, достигаемые через ненасыщенные дуги
        if (Link.mDest.mColor=CWhite)            // если вершина не помечена
           and (Link.mFlow < Link.mHigh) then begin  // и дуга не насыщена
          Link.mDirect:= +1;  // это увеличивающаяся дуга
          // помечаем концевую вершину и ставим её в очередь:
          with Link.mDest do begin
            mColor:= CGray;     // цвет концевой вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow); // поток
          end;
          Que.Put(Link.mDest); // в очередь приёмник дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины,
        // из которых поступает ненулевой поток
        if (Link.mOwner.mColor=CWhite)          // если вершина не помечена
           and (Link.mFlow > Link.mLow) then begin // и поток можно уменьшить
          Link.mDirect:= -1;  // это уменьшающая дуга
```

```
          // помечаем вершину (источник дуги) и ставим её в очередь:
          with Link.mOwner do begin
            mColor:= CGray;     // цвет исходной вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner); // в очередь источник дуги
        end;
        Link:= Node.InLinkNext;
      end;  // while
    end;  // while

    // Если сток не помечен, то увеличивающая цепочка не найдена
    // и приращение потока равно нулю
    if aDest.mColor = CWhite then Exit;
    // Здесь  увеличивающая цепочка найдена
    Result:= aDest.mFlow; // допустимое увеличение потока
    // Обратное движение от стока к истоку с корректировкой потока
    Node:= aDest;  // сток
    while Node <> aSource do begin
      with Node.mLink do Inc(mFlow, mDirect * Result); // mDirect = +1 / -1
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.CalcMaxFlow0 }
  Result:= 0;
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  Result:= CalcStartFlow; // начальный поток
  Que:= TBuffer.Create;   // рабочая очередь вершин
  // Повторяем поиск увеличивающих цепей, пока это возможно
  repeat
    Delta:= CalcDelta;    // подсчёт приращения потока
    Inc(Result, Delta);   // накопление результата
  until Delta=0;
  Que.Free;
end;
```

В начале процедуры вызывается локальная функция **CalcStartFlow**, подсчитывающая текущий суммарный поток, извергаемый источником. Здесь эта функция будет возвращать ноль, поскольку никакого начального потока не существует. Она заготовлена для следующей задачи, где поток ограничен в том числе снизу, и будет пояснена в главе 25.

Далее циклически вызывается локальная функция **CalcDelta**, строящая очередную увеличивающую цепочку и находящая максимальное приращение потока вдоль неё. Эта же функция изменяет текущий поток в дугах. Если увеличивающую цепь построить не удаётся, функция возвращает ноль и тогда алгоритм завершается.

Внутри локальной функции **CalcDelta** происходит следующее. Вначале очищаются данные в вершинах: цвет, обратные ссылки, а предполагаемый поток через вершину назначается «бесконечным». Затем красится и ставится в очередь обработки вершина-источник.

В цикле обработки очереди (расширении гаммы) из неё выбирается очередная вершина, и обрабатываются сначала исходящие (прямые) дуги, а затем входящие (обратные). Если *исходящая* дуга ведёт к неокрашенной вершине и поток в ней не насыщен, то дуга метится как прямая (**mDirect:=+1**), а в вершине-приёмнике этой дуги корректируются соответствующие данные: цвет, обратные ссылки и поток через вершину. Причём поток в вершине-приёмнике определяется как меньшее из двух: того потока, что дошёл до текущей вершины, и того, что может дополнительно пропустить исходящая дуга:

```
    mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow);
```

Схожим образом обрабатываем *входящие* дуги текущей вершины. Если входящая дуга принадлежит белой вершине, и поток в ней существует (точнее, это поток превышает минимальный), то дуга метится как обратная:

```
    mDirect:=-1;
```

а белая вершина красится, и в ней корректируются обратные ссылки и поток:

```
            mFlow:= Minimum(Node.mFlow, Link.mFlow);
```

Вновь окрашенные вершины ставятся в очередь, и цикл обработки очереди повторяется. Цикл завершается в двух случаях: 1) при извлечении из очереди вершины-стока (тут цепочка готова), либо 2) при опустошении очереди — тогда цепочка не может быть построена. В первом случае поле **mFlow** вершины-стока содержит то приращение потока, на которое можно увеличить его вдоль обнаруженной цепи. Тут следует завершающий аккорд: двигаясь обратно от *стока* к *истоку* по расставленным обратным ссылкам, наращиваем поток вдоль обнаруженной цепочки. Если же цепочка не обнаружена, возвращаем ноль.

24.9. Испытание

Ниже представлена программа для испытания метода поиска максимального потока для случая, когда нижняя граница для потока не задана.

Листинг 24-2 — Поиск максимального потока (нижняя граница не задана)

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';

var Gr : TGraph;       // граф
```

```
    S, T : TNode;      // вершины источника и стока
    Flow: integer;     // максимальный поток
begin
    Gr:= TGraphChars.Load(CFile);
    Gr.Expo;
    // Ввод данных о потоках
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('H');
    Flow:= Gr.CalcMaxFlow0(S, T);
    Writeln('Flow= ', Flow);
    Gr.ExpoLinksData;  // Вывод информации о потоках в дугах
    Readln;
    Gr.Free;
end.
```

Для числового примера взят граф, показанный на рис. 24-8, информация для его ввода представлена следующим файлом:

```
Test.txt – максимальный поток
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> B C
B -> E F
C -> D
D -> E
E -> H
F -> G
G -> H
H ->
FLOW:
AB= 0 5
AC= 0 4
BE= 0 2
BF= 0 4
CD= 0 4
DE= 0 4
EH= 0 3
FG= 0 4
GH= 0 4
```

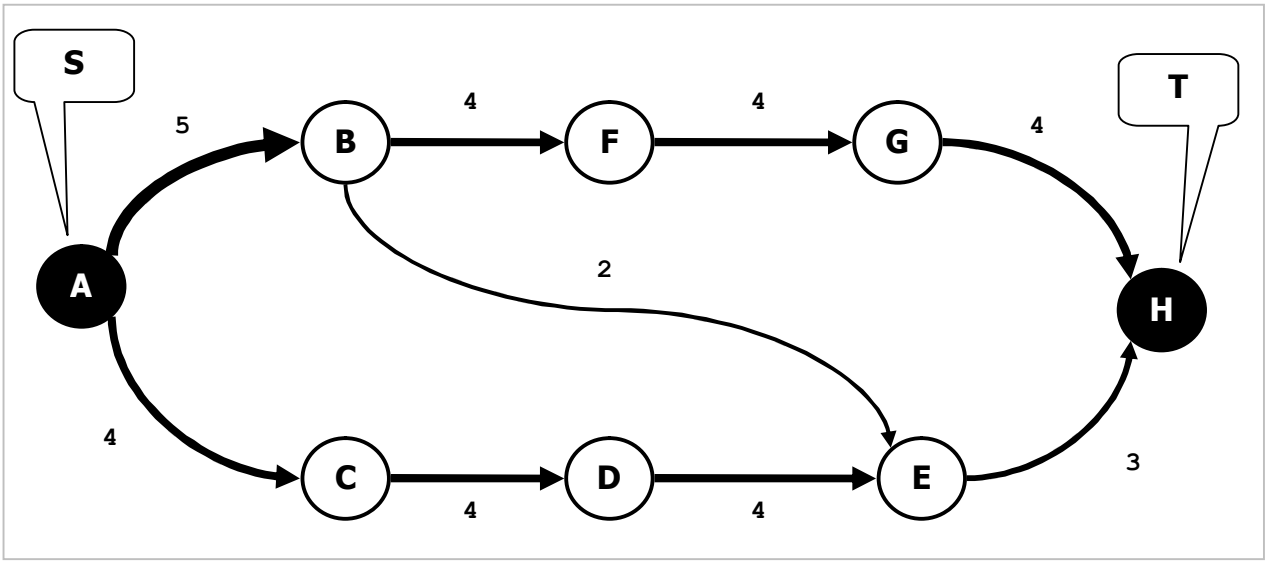


Рис. 24-8 — Граф для тестирования

Числа рядом с дугами показывают их пропускную способность. В ходе испытания метода, были последовательно найдены четыре увеличивающие цепочки, показанные на следующих рисунках.

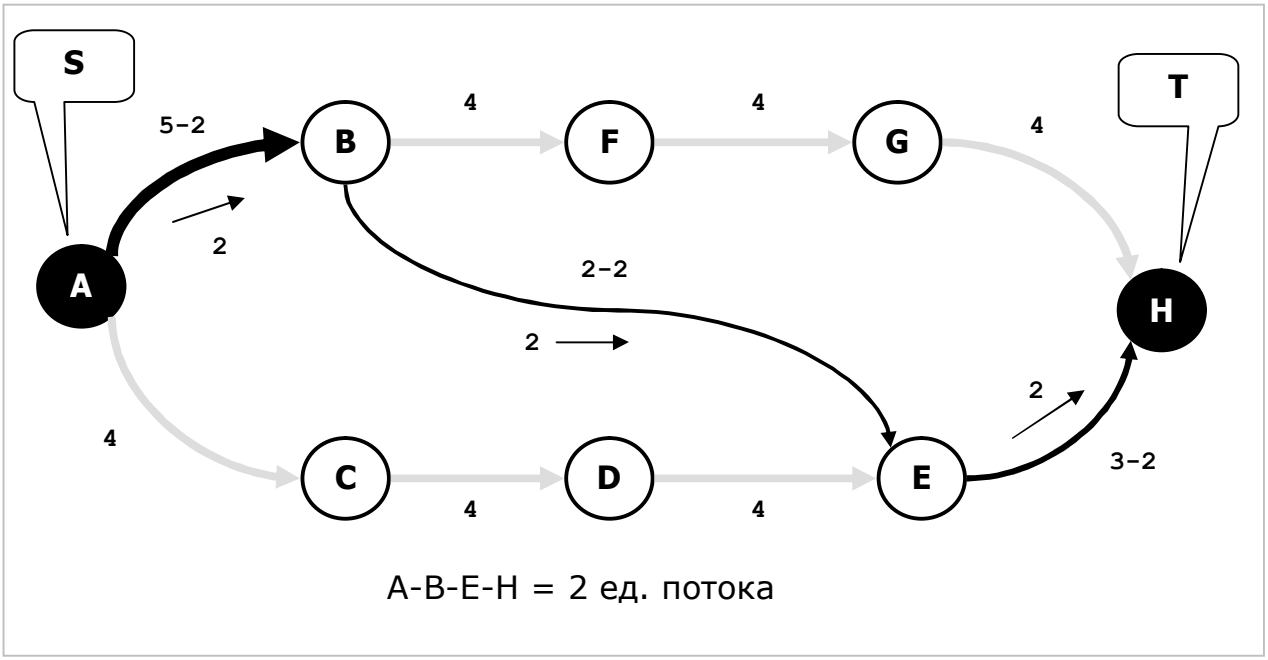


Рис. 24-9 — Пересылка двух единиц потока по цепочке A-B-E-H, насыщается дуга BE

В первой из обнаруженных цепочек *A-B-E-H* насыщается прямая дуга *B-E*. Стрелки показывают направление и величину потока. Текущий запас пропускной способности дуг представлен разностями между пределом пропускной способности и текущим потоком.

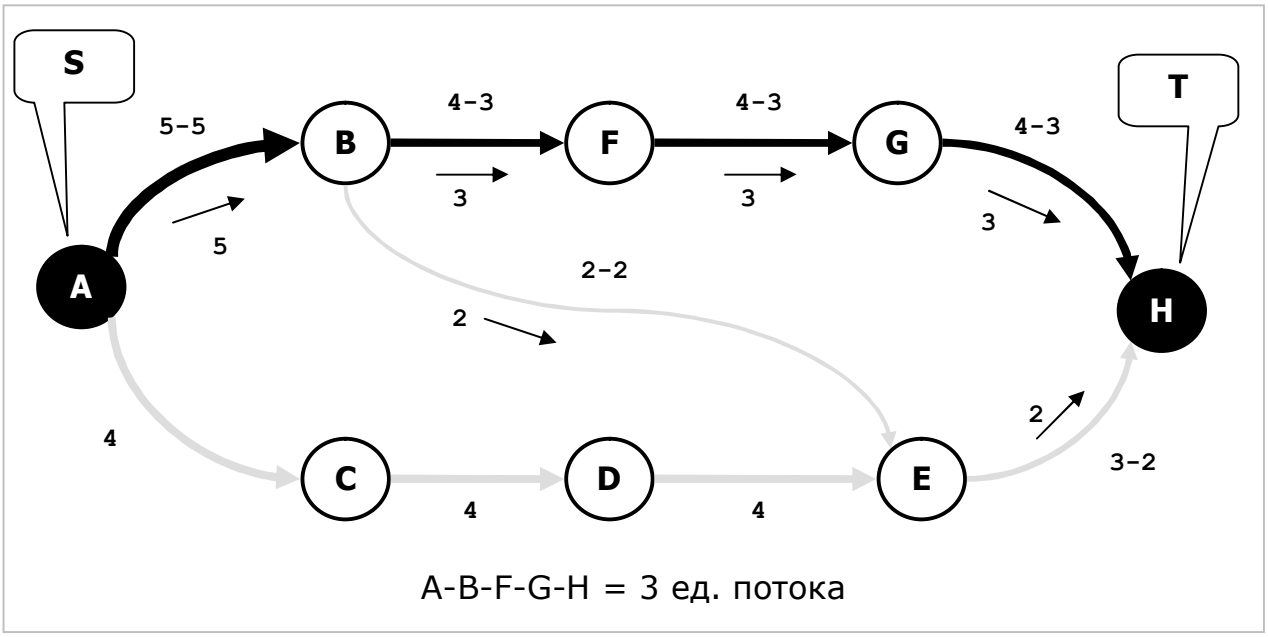


Рис. 24-10 — По цепочке A-B-F-G-H следуют три единицы потока, насыщается дуга A-B

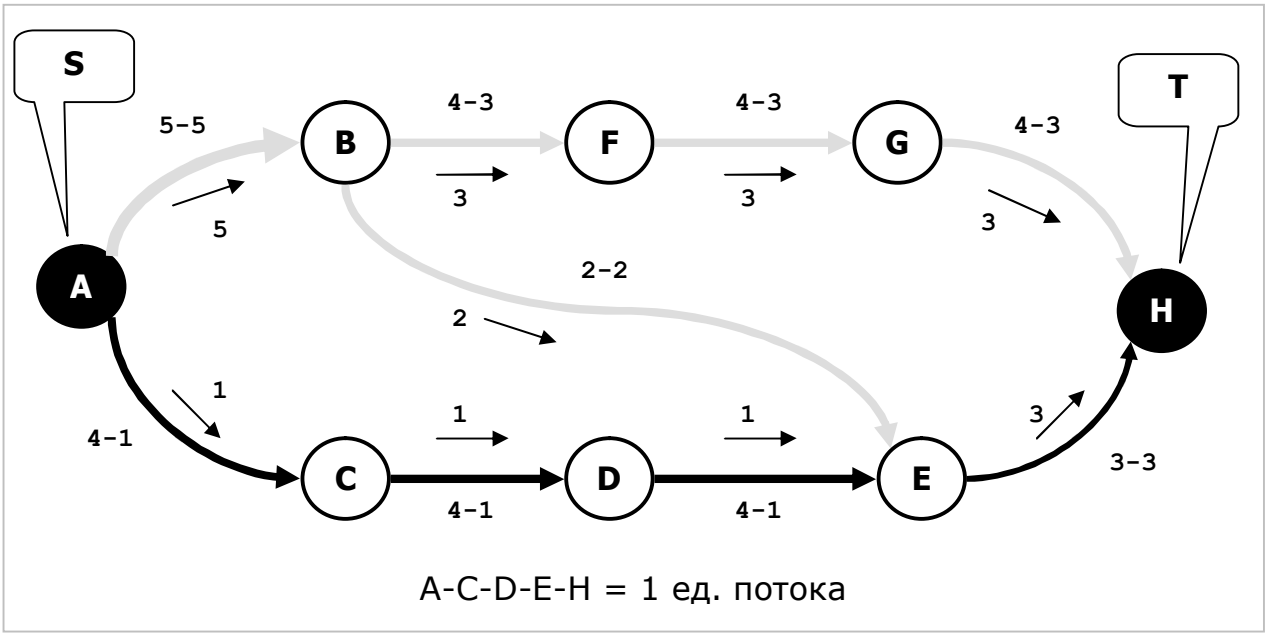


Рис. 24-11 — По цепочке A-C-D-E-H следуют три единицы потока, насыщается дуга E-H

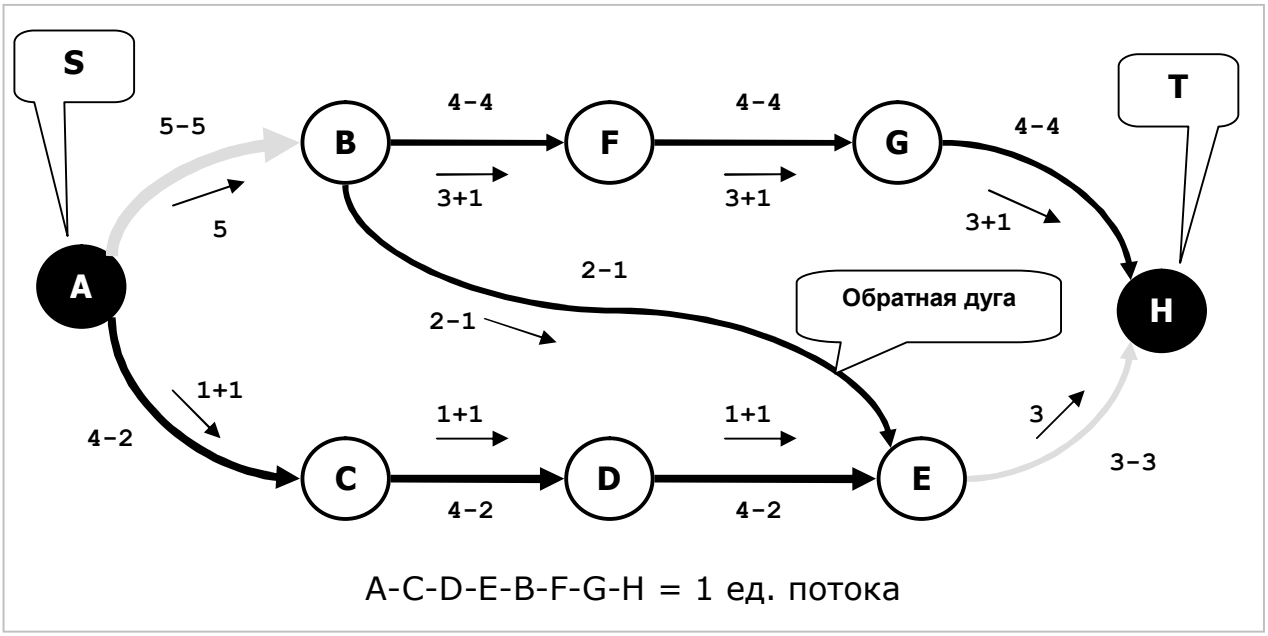


Рис. 24-12 — По цепочке A-C-D-E-B-F-G-H следуют одна единица потока, насыщаются дуги B-F, F-G, G-H

Обратим внимание на дугу *B-E*. В первой из обнаруженных цепочек (рис. 24-9) она была прямой, а в последней (рис. 24-12) стала обратной. Уменьшение оттока в дуге *B-E* привело к увеличению потока вдоль цепочки, но как он перераспределился? Одна единица, исходящая из вершины *B*, переориентировалась в цепочку *F-G-H*. При этом недостаток этой единицы потока в вершине *E* был компенсирован потоком из цепи *A-C-D*.

Итак, после четырёх итераций был получен максимально возможный поток в данной сети, составляющий **7** единиц.

24.10. Итоги

24.10.1. Сетью называют ориентированный граф, дугам которого приписаны пропускные способности. Посредством сетей моделируют транспортировку материальных или информационных объектов.

24.10.2. Основная потоковая задача состоит в поиске максимальной пропускной способности сети между истоком и стоком, и распределении потока по дугам графа с тем, чтобы достичь этого максимума.

24.10.3. Принцип решения задачи состоит в поиске кратчайших увеличивающих цепей между источником и стоком с наращиванием потока вдоль этих цепей.

24.11. Задания

24.12. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 91 (пункт 4.2)
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 25
Поток, ограниченный сверху и снизу

В предыдущей главе мы решили базовую потоковую задачу: нашли *максимальный* поток в сети. Теперь рассмотрим случай, когда для некоторых или всех дуг графа поток ограничен не только *сверху*, но и *снизу* (задан нижний предел для потока). Задачи этого рода порой не имеют решения, вот пример.

Пусть N пассажиров следуют из пункта S в пункт T, с пересадкой их с автобуса на автобус в промежуточном пункте X (рис. 25-1).

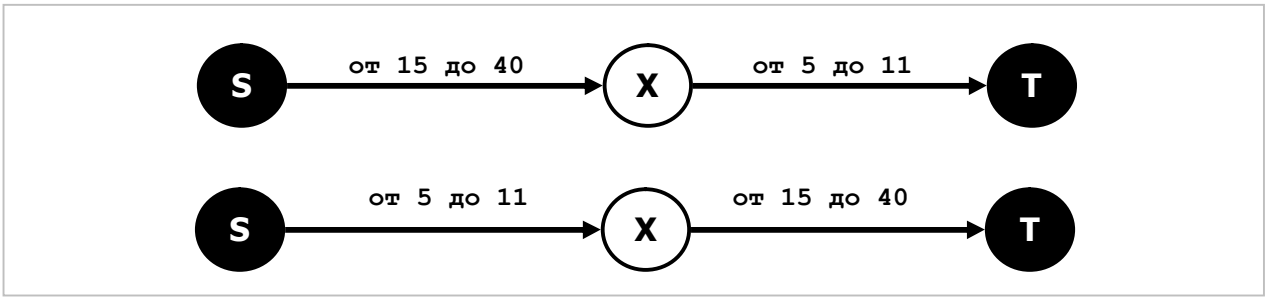


Рис. 25-1 — Примеры неосуществимых перевозок

Один из автобусов — крупный, и вмещает до 40 пассажиров, а другой — небольшой, вмещает до 11 пассажиров. Кажется, что перевозка не более чем 11 пассажиров двумя рейсами вполне осуществима. Однако на беду в дело вмешались деньги: владельцу крупных автобусов не выгодны рейсы низкой наполняемости, и он наотрез отказывается везти менее 15 пассажиров. На рис. 25-1 показаны нижний и верхний пределы наполнения приемлемых рейсов. Ясно, что в пункте пересадки все или часть пассажиров застрянет: либо, не вместившись в малый автобус, либо не уехав на большом из-за нерентабельности рейса.

Отсюда видим, что далеко не всякое сочетание минимальных и максимальных пропускных способностей дуг даёт решение потоковой задачи. И потому поиск *максимального* потока надо предварить поиском *минимального* потока в сети (и соответствующего распределения минимального потока по дугам). Если пропуск минимального потока осуществим, то максимальный поток находится уже известным алгоритмом, а иначе задача не имеет решения.

25.1. Эквивалентная схема

На рис. 25-2 представлен граф, для некоторых дуг которого указаны — через двоеточие, — помимо верхнего, ещё и нижний ненулевой предел для потока.

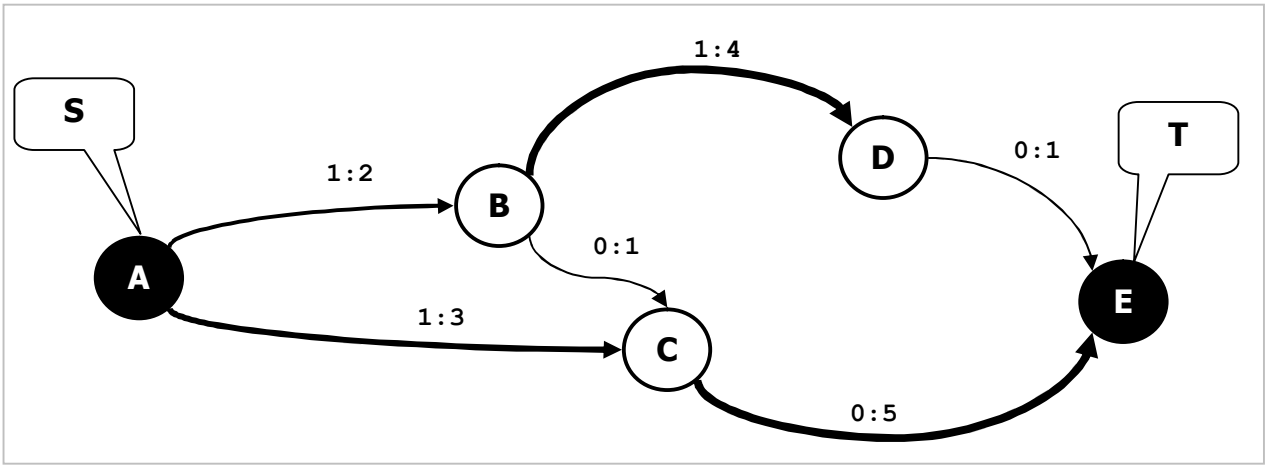


Рис. 25-2 — Граф с нижним и верхним ограничениями потока

Первая из поставленных задач — поиск минимального потока. Она будет (неожиданно!) решена через поиск максимального потока. Для этого из исходного графа будет создан его вспомогательный эквивалент. Суть идеи в том, чтобы отделить собственный минимальный поток каждой дуги от минимальных потоков в других дугах. Рассмотрим дугу графа *X-Y* с нижней **L** и верхней **H** границами пропускной способности (рис. 25-3).

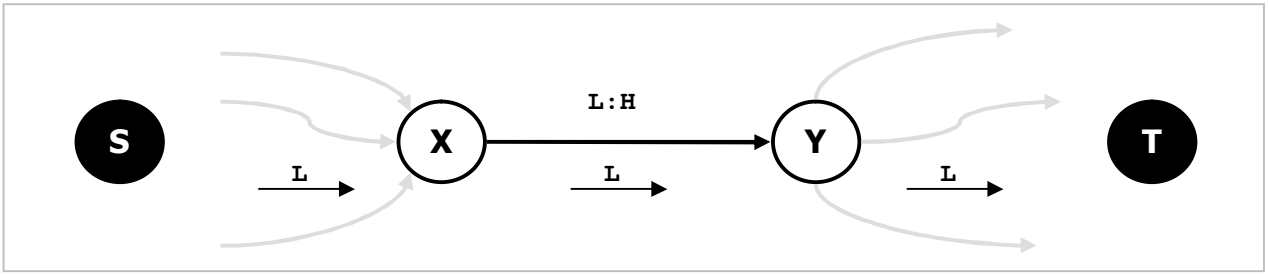


Рис. 25-3 — Дуга исходного графа с нижним (L) и верхним (H) пределами пропускной способности

Резерв **D = H−L** можно трактовать как верхний предел пропускной способности дуги, используемый для пропуска минимальных потоков *других* дуг. Разумно предположить, что если минимальный поток в сети осуществим, то через каждую дугу должен протекать, по меньшей мере, её минимальный поток **L**, или несколько больший в пределах её пропускной способности **H**. Этот поток **L** должен сначала пройти от источника *S* к началу дуги, а затем от конца дуги к стоку *T*, используя *резервы* **D** в других дугах. Если такое продвижение потока осуществимо, то минимальный поток в графе возможен, и его можно найти.

Для отделения «своего» потока от потоков «чужих» дуг, направим «свой» минимальный поток в «кругосветное путешествие» по чужим дугам, а также через сток *T* и исток *S* сети. Для этого учредим вспомогательные исток *Sx* и сток *Tx* (см. рис. 25-4).

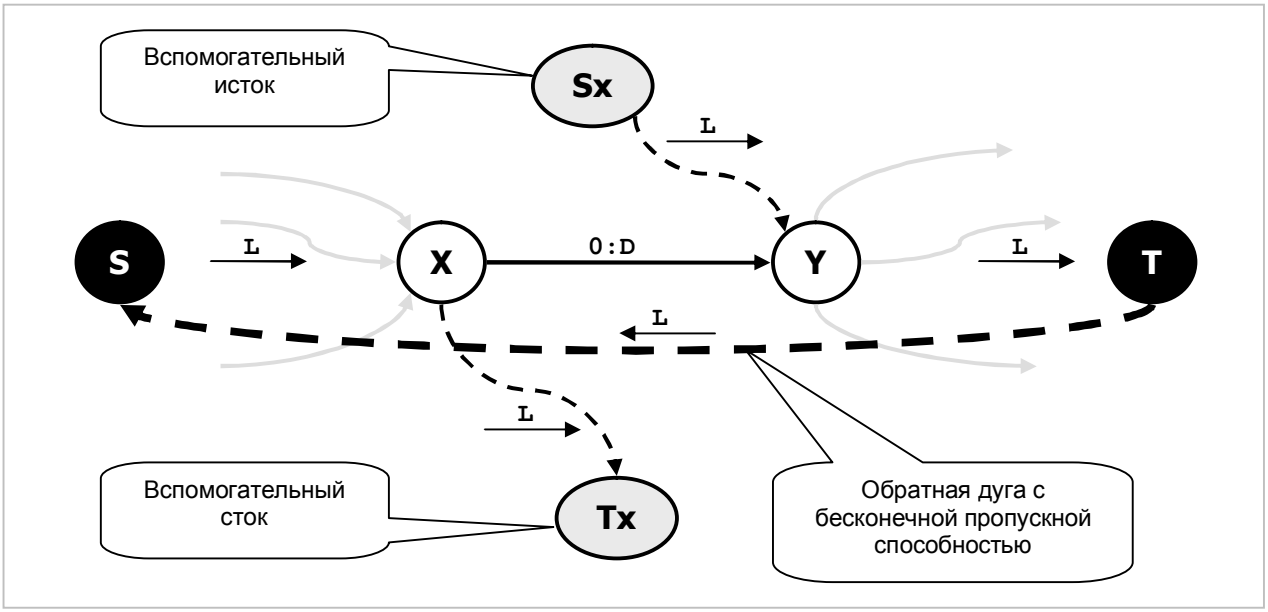


Рис. 25-4 — «Кругосветное путешествие» минимального потока L дуги X

На рис. 25-4 показаны элементы, введённые в сеть для отделения собственного потока дуги от «чужих» потоков, и организации «кругосветки» минимальных потоков. Во-первых, сток сети соединён с истоком обратной дугой *T-S* с бесконечной пропускной способностью, и теперь всё, что приходит в сток *T*, свободно стекает обратно в исток *S*. Во-вторых, введены две вспомогательные вершины: исток *Sx* и сток *Tx* (прежние исток и сток временно становятся транзитными вершинами). Теперь собственный минимальный поток дуги течёт не по ней, а исходит из вспомогательного истока *Sx*, и далее следует по цепочке:

Sx → Y ...→... T □ S ...→... X □ Tx

Избавив дуги графа от собственных минимальных потоков *L*, мы вправе теперь снизить их пропускную способность на ту же величину *L*, и тогда их *минимальная* пропускная способность обратится в ноль, а *максимальная* составит разность **D=H-L**. Это третье изменение в исходном графе.

Разделив потоки всех дуг, мы получим новый граф с другими источником и стоком и — главное! — с дугами, в которых *минимальная* пропускная способность равна нулю. Так задача о *минимальном* потоке свелась к уже решённой задаче о *максимальном* потоке. Назначим вновь введённым вспомогательным дугам (они указаны пунктиром) максимальные пропускные способности **L**, и построим максимальный поток от источника *Sx* к стоку *Tx*. Затем проверим насыщенность вспомогательных дуг (сравним сумму их пропускных способностей с суммой их потоков). Если вспомогательные дуги окажутся насыщенными, следовательно, все минимальные потоки успешно преодолели «кругосветку», наполнив каждую дугу минимальным потоком для данной сети. Иное будет означать неспособность сети пропустить минимальный поток.

25.2. Дело техники

Теперь обратимся к техническим деталям, и разберёмся с вспомогательными дугами. По нашей задумке, каждую вершину надо связать со вспомогательным истоком и стоком таким количеством дуг, которое равно количеству входящих и исходящих дуг у этой вершины. Однако простой граф не предусматривает между вершинами более одной дуги того же направления. Проблема решается просто: совокупность из нескольких параллельных дуг заменим одной дугой с суммарной пропускной способностью. Мало того: пару дуг (от источника к вершине, и от вершины к стоку) либо заменим одной дугой, либо вообще отбросим, — поясним это на числовом примере и следующих рисунках.

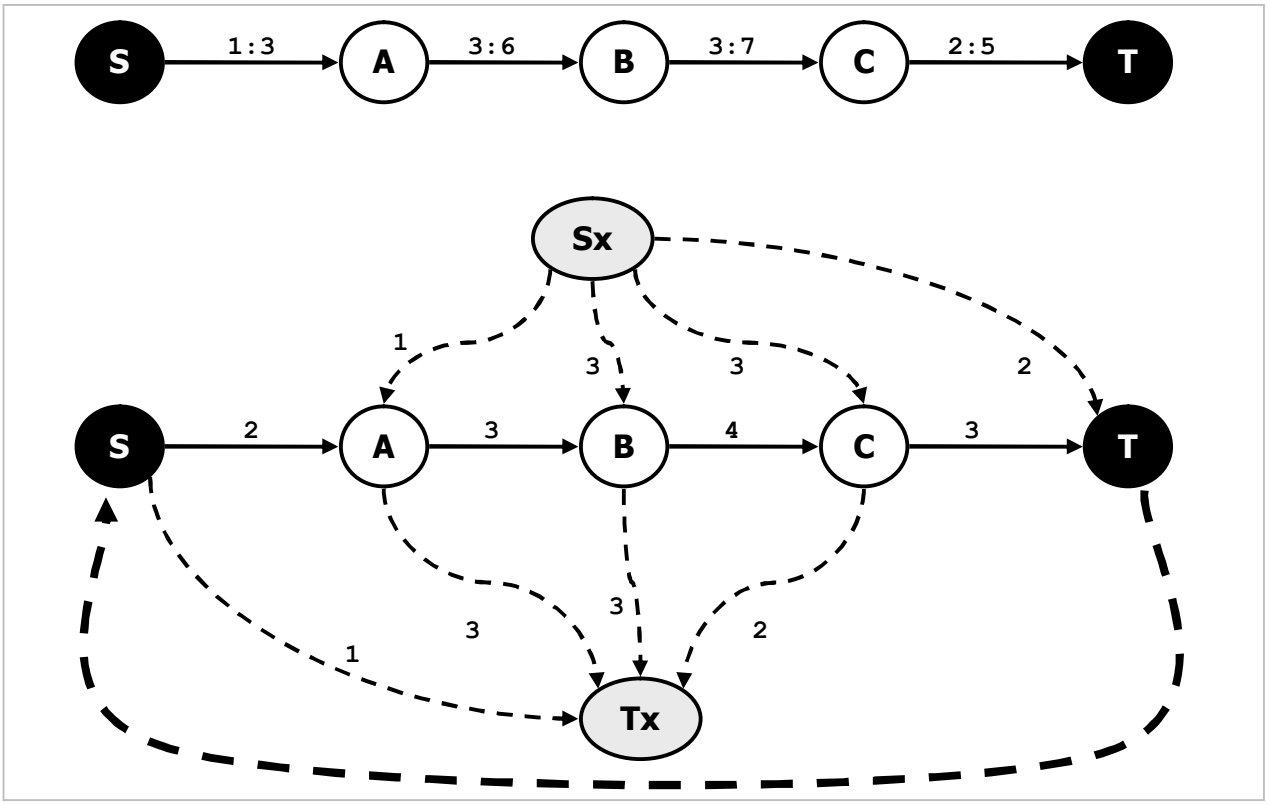


Рис. 25-5 — Исходная и вспомогательная сети

На рис. 25-5 показана исходная сеть, где для всех дуг заданы минимальные и максимальны пределы, а ниже — соответствующая ей вспомогательная сеть. Очевидно, что часть потока пройдёт в ней от *Sx* до *Tx* кратчайшими маршрутами, минуя исходные дуги графа и обратную дугу. Стало быть, эту часть потока можно без ущерба отбросить, поскольку она не влияет на результат. Так, можно удалить дуги *Sx→B* и *B→Tx*. А вместо пары дуг *Sx→A* и *A→Tx* оставить дугу *A→Tx* с пропускной способностью **3-1=2**. Аналогично пару дуг *Sx→C* и *C→Tx* заменим одной дугой *Sx→C* с пропускной способностью **3-2=1**. Результат этих сокращений показан на рис. 25-6.

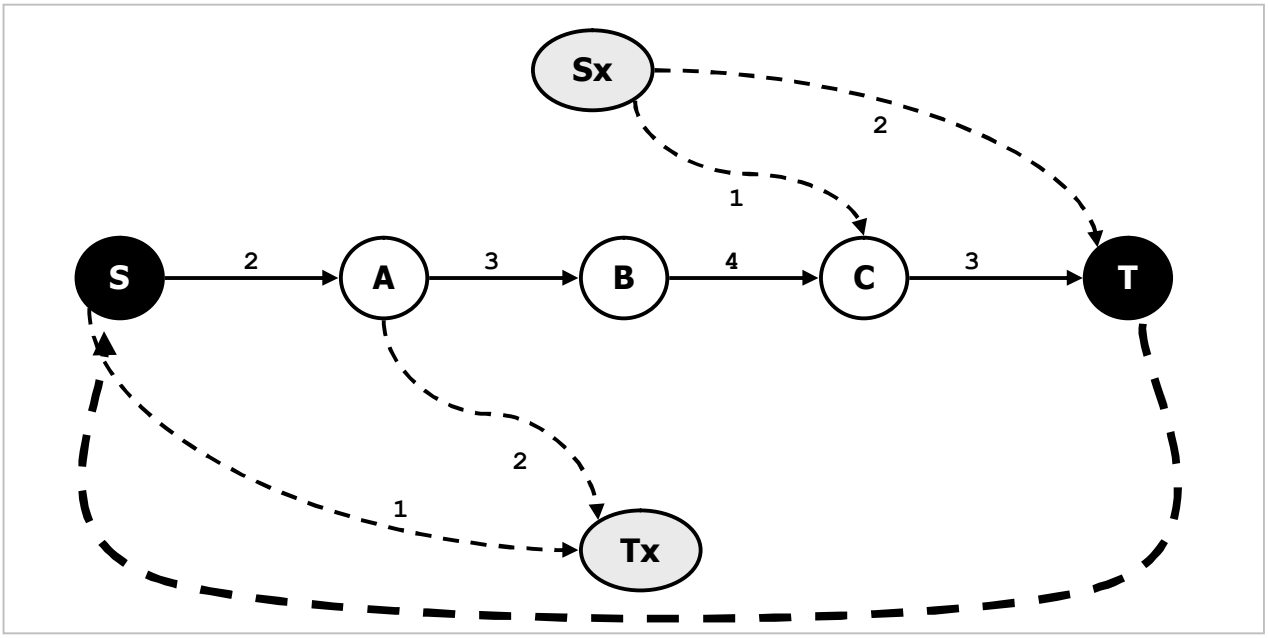


Рис. 25-6 — Сокращённая вспомогательная сеть

25.3. Алгоритм

Подведём промежуточный итог рассуждений и дадим краткое описание алгоритма поиска *минимального* потока в сети:

- Сток сети соединим с истоком обратной дугой с неограниченной пропускной способностью.
- Введём в сеть две вспомогательные вершины: исток *Sx* и сток *Tx*.
- Для каждой вершины определим разность между суммой «собственных» потоков её исходящих дуг и той же суммой входящих. Если эта разность больше нуля, то соединим вершину с вспомогательным стоком *Tx* дугой с пропускной способностью, равной этой разности. Если разность отрицательна, то соединим вспомогательный исток *Sx* с этой вершиной дугой с пропускной способностью, равной этой разности по модулю. При нулевой разности никаких дополнительных дуг не создаём.
- Нижний предел пропускной способности всех исходных дуг графа назначим равным нулю, а верхний — разности **`D=H-L`**.
- Строим максимальный поток между вспомогательным истоком и стоком.
- Исследуем насыщенность вспомогательных дуг. Если все они насыщены, то минимальный поток возможен, и поток, распределённый в исходных дугах, ему соответствует, а иначе поток невозможен.
- Удалим все вспомогательные дуги и вершины.

Метод вычисления минимального потока представлен ниже.

Листинг 25-1 — Вычисление минимального потока

```
function TGraph.CalcMinFlow(aSource, aDest: TNode): integer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Восстановление данных о потоке после удаления искусственных вершин и дуг
  // arg = false -- минимальный поток не существует
  // arg = true  -- минимальный поток найден

  procedure RestoreFlowData(arg: boolean);
  var Node: TNode;
      Link: TLink;      // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Перебор исходящих линков вершины:
        Link:= OutLinkFirst;
        while Assigned(Link) do begin
          with Link do begin
            mLow:= mTemp;           // минимум
            mHigh:= mHigh + mLow;   // максимум
            // текущий минимальный поток:
            if arg
              then mFlow:= mFlow + mLow  // в случае успеха
              else mFlow:= 0;            // если поток не существует
          end; // with
          Link:= OutLinkNext;
        end; // while
      end; // with
      Node:= NodeNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Node: TNode;       // текущая вершина
    Link: TLink;       // текущий линк
    Reverse: TLink;    // обратный линк T->S во вспомогательном графе
    Flow: integer;     // максимальный поток
    SumLow: integer;   // сумма всех минимумов
    Sx: TNode;         // вспомогательный исток
    Tx: TNode;         // вспомогательный сток
    OldLink: TLink;    // линк T->S в исходном графе
    OldLow, OldHigh: integer; // сохранённые данные обратного линка

begin    { TGraph.CalcMinFlow }

  // Если не указаны вершины, то выход
  if not Assigned(aSource) or not Assigned(aDest) then begin
    Result:=-1;
    Exit;
  end;

  // Создаём:
  Sx:= TNode.Create(0, Self); // вспомогательный исток
  Tx:= TNode.Create(0, Self); // вспомогательный сток
  SumLow:=0;    // здесь накапливаем сумму всех минимальных потоков
  // Первый перебор вершин с подсчётом сумм исходящих минимальных потоков:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mFlow:=0; // сумма исходящих минимумов
    with Node do begin
```

```
    // Перебор исходящих линков вершины:
    Link:= OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        Inc(Node.mFlow, mLow);  // сумма исходящих минимумов
        mHigh:= mHigh - mLow;   // пересчёт максимума
        mTemp:= mLow;           // временно сохраняем минимум
        mLow:= 0;               // и обнуляем его
        mFlow:=0;               // обнуляем поток
      end;
      Link:= OutLinkNext;
    end;
  end; // with
  Node:= NodeNext;
end; // while

// Второй перебор вершин с подсчётом сумм входящих минимальных потоков:
Node:= NodeFirst;
while Assigned(Node) do begin
  with Node do begin
    // Перебор входящих линков вершины:
    Link:= InLinkFirst;
    while Assigned(Link) do begin
      Dec(Node.mFlow, Link.mTemp); // минус сумма исходящих минимумов
      Link:= InLinkNext;
    end;
  end;
  // Если вершина является источником или стоком:
  if Node.mFlow <> 0 then begin
    // Если вершиная является источником:
    if Node.mFlow > 0 then begin
      // создаём дугу из вершины во вспомогательный сток
      Link:= Node.MakeLink(Tx, 0);  // создаём линк
      Link.mHigh:= Node.mFlow;      // с данными о потоке
    end else begin
      // создаём дугу из вспомогательного истока в вершину
      Link:= Sx.MakeLink(Node, 0);  // создаём линк
      Link.mHigh:= -Node.mFlow;      // с данными о потоке
    end
  end;
  Inc(SumLow, Abs(Node.mFlow)); // двойная сумма всех минимальных потоков
  Node:= NodeNext;
end; // while
SumLow:= SumLow div 2;  // одинарная сумма всех минимальных потоков

// Если сумма минимальных потоков = 0, то выходим из процедуры
if SumLow=0 then begin
  // Освобождаем:
  Sx.Free;     // вспомогательный исток
  Tx.Free;     // вспомогательный сток
  // Восстанавливаем данные о потоках:
  RestoreFlowData(false);
  Result:=0;
  Exit;
end;

// Вставляем вспомогательные вершины в граф:
mNodes.Insert(Sx);   // исток
mNodes.Insert(Tx);   // сток

// Соединяем обратной дугой сток с истоком
// (поток в обратной дуге не ограничен):
```

```
  // На случай отсутствия обратной дуги:
  OldLow:= 0; OldHigh:=0;

  // Обратная дуга уже существует?
  OldLink:= aDest.GetLink(aSource);
  if Assigned(OldLink) then begin
    // да, назначаем её в качестве реверсивной и запоминаем параметры
    Reverse:= OldLink;    // реверсивная дуга
    with OldLink do begin
      // да, запоминаем данные потока
      OldLow := mLow;     // нижний предел
      OldHigh:= mHigh;    // верхний предел
      // и устанавливаем новые
      mLow := 0;          // нижний предел отсутствует
      mHigh:= MaxInt;     // неограниченная пропускная способность
    end // with
  end else begin
    // здесь обратной дуги нет, поэтому создаём новую дугу
    Reverse:= aDest.MakeLink(aSource, 0); // дуга
    Reverse.mHigh:= MaxInt;   // с максимальной пропускной способностью
  end;

  // Во вновь построенном вспомогательном графе
  // вычисляем максимальный поток между вспомогательными истоком и стоком

  Flow:= CalcMaxFlow0(Sx, Tx);

  // После этого минимальный поток протекает через реверсную дугу:
  Result:= Reverse.mFlow;  // запоминаем
  Reverse.mFlow:= 0;       // и обнуляем

  // Удаляем обратную дугу, или восстанавливаем предыдущую, если она была

  // Обратная дуга существовала?
  if Assigned(OldLink) then with OldLink do begin
    // да восстанавливаем
    mLow := OldLow;
    mHigh:= OldHigh;
  end else begin
    // Дуга в исходном графе не существовала,
    // разрываем обратную связь между стоком и истоком
    aDest.RemoveLink(aSource);
  end;

  // Удаляем из графа и освобождаем:
  RemoveNode(Sx);  Sx.Free;     // вспомогательный исток
  RemoveNode(Tx);  Tx.Free;     // вспомогательный сток

  // Если вспомогательный максимальный поток (Sum)
  // равен сумме минимальных потоков (SumLow)
  // то минимальный поток в исходном графе существует

  if Flow = SumLow then begin
    // вспомогательные дуги насыщены
    RestoreFlowData(true); // формируем новые данные о потоках
  end else begin
    // вспомогательные дуги НЕ насыщены
    Result:=-1;            // минимального потока не существует
    RestoreFlowData(false); // восстанавливаем прежние данные о потоках
  end;
end;
```

Несколько слов об обратной дуге. В листинге предполагается, что в исходной сети такая дуга может как существовать, так и не существовать. В первом случае вначале запоминаются параметры этой дуги, а перед выходом из метода они восстановливаютмя.

25.4. Метод вычисления максимального потока

Теперь рассмотрим поиск *максимального* потока в сети для случая, когда заданы *ненулевые* нижние границы. Мы помним, что этот поток при *нулевых* нижних границах вычислялся последовательным наращиванием потока вдоль увеличивающих цепей, начиная с *нулевого* потока. Здесь поступим точно также, только начальным будет не нулевой, *минимальный* поток.

Листинг 25-2 — Вычисление максимального потока для ненулевых нижних границ

```
function TGraph.CalcMaxFlow(aSource, aDest: TNode): integer;
begin
  // Сначала вычисляем минимальный поток:
  Result:= CalcMinFlow(aSource, aDest);
  // Если он не существует, то не существует и максимального:
  if Result < 0 then Exit;
  // После формирования минимального потока формируем максимальный:
  Result:= CalcMaxFlow0(aSource, aDest);
end;
```

Здесь уместно вернуться к роли локальной функции **`CalcStartFlow`**, расположенной в методе **`CalcMaxFlow0`**, и напомнить, что именно она вычисляет начальное значение потока, текущего по дугам сети при старте алгоритма.

25.5. Испытание

Для испытаний методов вычисления минимальных и максимальных потоков воспользуемся следующей программой.

Листинг 25-3 — Программа для испытания потоковых методов

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';
var Gr : TGraph;
    S, T : TNode;
    Flow: integer;
begin
    Gr:= TGraphChars.Load(CFile);
```

```
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('E');
    Flow:= Gr.CalcMinFlow(S, T);
    Writeln('MinFlow= ', Flow);
    Gr.ExpoLinksData;
    Writeln('=================');
    Flow:= Gr.CalcMaxFlow(S, T);
    Writeln('MaxFlow= ', Flow);
    Gr.ExpoLinksData;
    Readln;
    Gr.Free;
end.
```

Ниже представлены данные для ввода испытательного графа и сам этот граф.

```
Test.txt – минимальный и максимальный поток
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
0 - дуги (1 = нагруженные)
5 - количество вершин
A B C D E
A -> B C
B -> C D
C -> E
D -> E
E ->
FLOW:
AB= 1 2
AC= 1 3
BC= 0 1
BD= 1 4
CE= 0 5
DE= 0 1
```

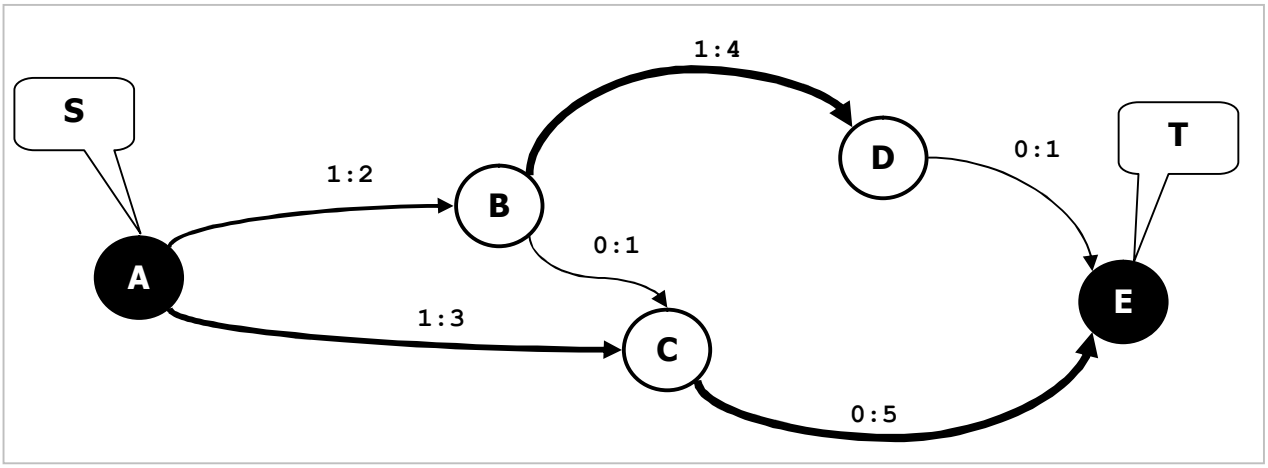


Рис. 25-7 — Граф с ненулевыми нижними границами

На рис. 25-1 представлена вспомогательная сеть, максимальный поток в которой составляет 2 единицы, а вспомогательные дуги насыщены.

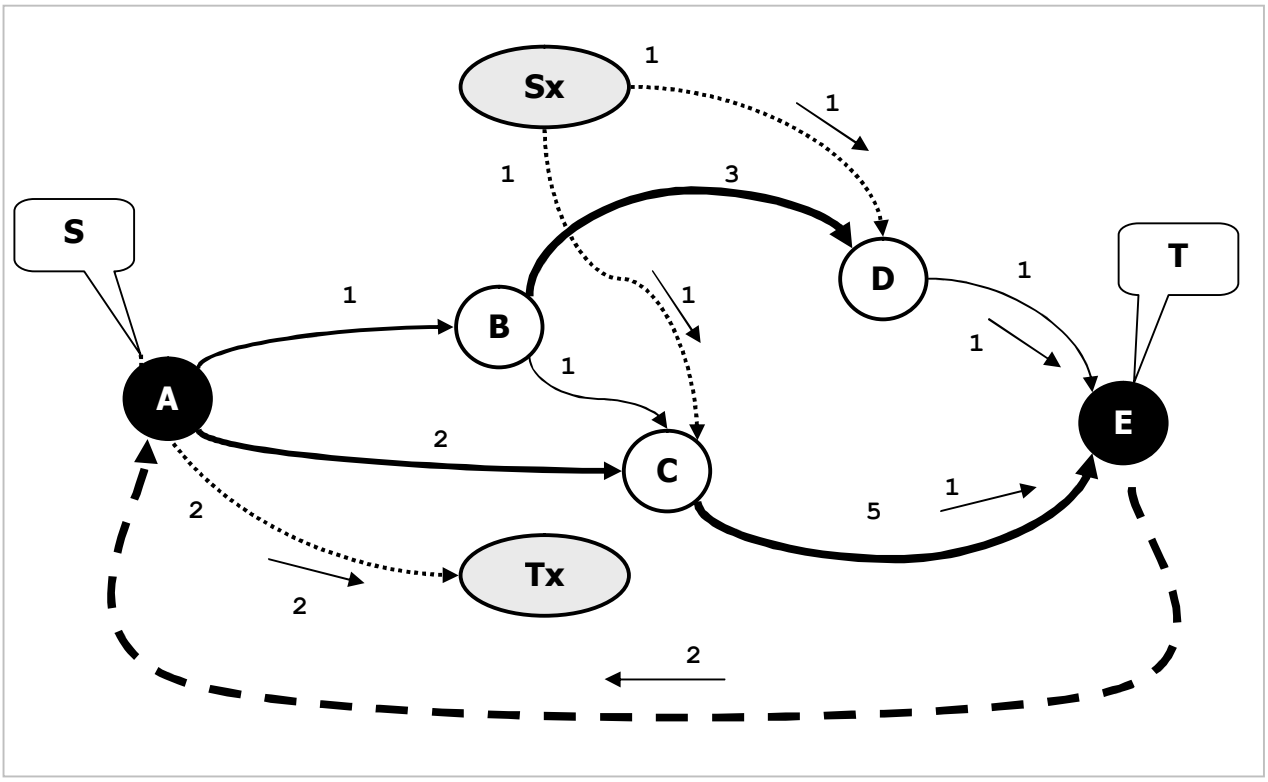


**Рис. 25-8 — Вспомогательная сеть,
максимальный поток в которой составляет 2 единицы**

На двух следующих рисунках дано распределение минимального и максимального потока по дугам графа.

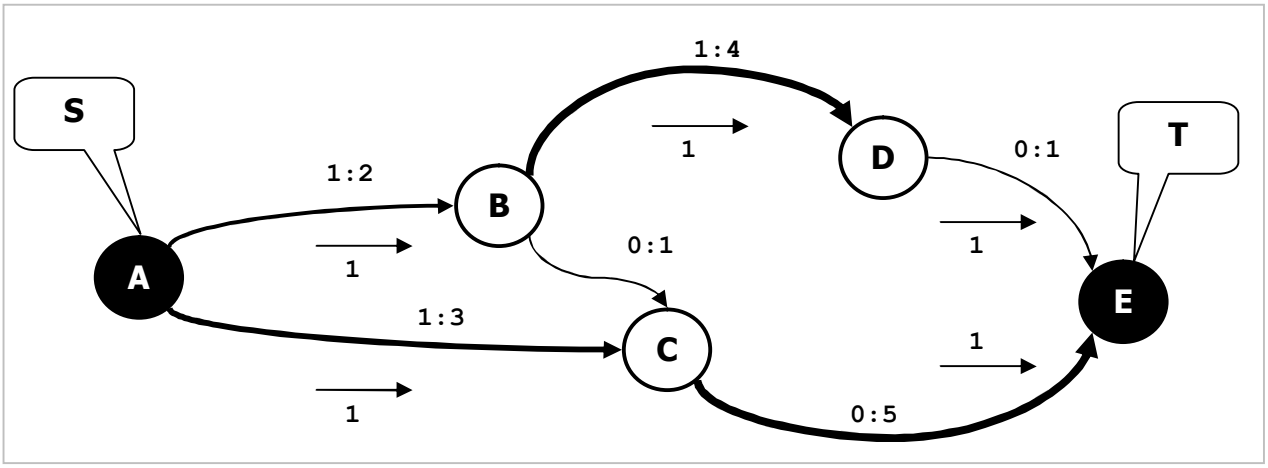


Рис. 25-9 — Минимальный поток в сети составляет 2 единицы

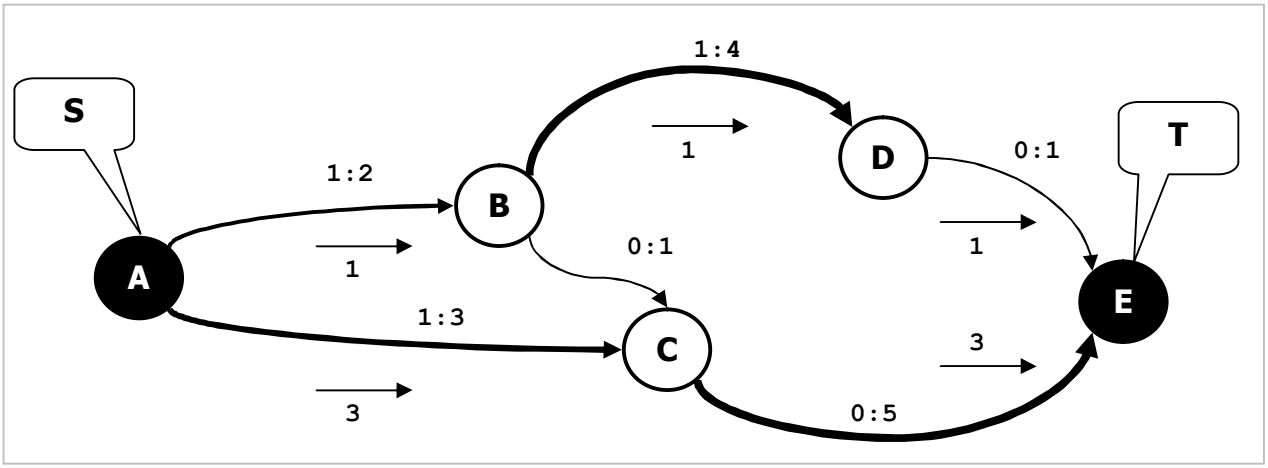


Рис. 25-10 — Максимальный поток в сети составляет 4 единицы

25.6. Итоги

25.6.1. Ограничение на нижнюю и верхнюю пропускную способность дуг может повлечь невозможность пропуска потока в сети.

25.6.2. Для поиска минимального потока в сети собственные минимальные потоки дуг отделяются от «чужих» минимальных потоков и соответственно уменьшаются их пропускные способности. В результате вспомогательных преобразований получается новая сеть, на которой решается задача поиска максимального потока. Так поиск минимального потока сводится к поиску максимального.

25.6.3. Нахождение минимального потока в сети (если он возможен), выполняется уже известным способом поиска максимального потока.

25.7. Задания

25.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
√ 2	Басакер Р. Саати Т.	Конечные графы и сети	стр. 327 (7.8)
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
7	Кристофидес Н.	Теория графов. Алгоритмический подход	
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 26
Минимальная стоимость потока

Вспомним о потоке машин, текущем по городским улицам (глава 24). Положим, нам известно количество автомобилей, проезжающих город в единицу времени (поток), а также длина всех участков улиц между перекрёстками. Пусть расходы на проезд каждой машины пропорциональны длине этих улиц (дуг). Тогда возникают два вопроса: 1) как разумней распределить поток по улицам с целью его удешевления? и 2) какова будет минимальная стоимость этого потока?

26.1. Формулировка задачи

Итак, задача о *минимальной стоимости потока* формулируется следующим образом. Дана сеть, где каждой дуге приписана пропускная способность и *цена* пересылки по ней единицы потока, а также *общая* величина протекающего потока. Надо найти минимальную *стоимость* пересылки этого потока и соответствующее *распределение* потока по дугам сети.

Эту задачу называют ещё задачей о потоке *минимальной стоимости*, хотя, с другой стороны, поток минимальной стоимости — это нулевой поток, стоимость которого равна нулю.

26.2. Алгоритм

Принцип решения этой задачи схож с идеей поиска максимального потока (см. главу 24). Напомню, что там мы последовательно находили увеличивающие цепочки дуг между истоком и стоком сети, начиная с кратчайшей цепи, и трактовали там длину цепи как число дуг, её составляющих. Здесь поступим аналогично, но будем последовательно отыскивать наиболее *дешёвые* цепи. Стоимость цепи равна сумме цен составляющих её дуг (для цены дуги назначено поле **`TLink.mValue`**). В целом алгоритм решения задачи таков:

1. Начальный поток **`F`** и стоимость его транспортировки **`C`** принимаем равными нулю (**`F=0`**, **`C=0`**) .
2. «Покупаем», по крайней мере, одну, наименее дорогую увеличивающую цепочку в сети, и находим цену пересылки по ней единицы потока **`C1`**. Если ни одной цепи «купить» не удаётся, то возвращаем отрицательный результат — требуемый поток недостижим.
3. Находим допустимое приращение потока, текущего вдоль очередной «купленной» цепи по цене **`C1`**. Если приращение потока невозможно, переходим к пункту 2, а иначе — к пункту 4.
4. Наращиваем поток **`F`** и стоимость его транспортировки **`C`**, одновременно распределяя поток по дугам. Если поток достиг нужной величины, то успешно завершаем алгоритм, а иначе переходим к пункту 3.

Итак, мы начинаем с «покупки» хотя бы одной наименее дорогой увеличивающей цепи (или сразу несколько равноценных). При отсутствии таких цепей задача не решается. Затем находим одно или несколько приращений потока, и корректируем соответственно и *ПОТОК*, и его *СТОИМОСТЬ*. Когда после приращения потока он достигнет нужного значения, завершаем процесс, в противном случае «покупаем» следующую по стоимости цепь. Забегая вперёд, отмечу, что определять приращение потока согласно п.3 мы будем почти так же, как делали это в задаче поиска максимального потока, и потому главное внимание обратим на «покупку» цепочек.

26.3. Торг уместен

Слово «покупка» взято в кавычки по той причине, что под покупкой понимается цена, выплачиваемая за пересылку единицы потока от истока к некоторой точке сети — промежуточной вершине, либо к стоку. С другой стороны, эту цену можно зрительно изобразить как радиус круга, очерченного из истока *S*. Тогда все дуги (или части дуг), попавшие внутрь условного круга, считаются приобретёнными. К примеру, в этом круге может оказаться две или более увеличивающие цепи с одинаковой ценой **C**, и тогда по этой цене приобретаются сразу все эти цепочки. Стоимость проходящего по ним потока будет равна величине суммарного потока в цепях, умноженной на **C** — цену одной цепочки.

Приняв это условное понимание покупки (далее упоминаем её без кавычек), обратимся теперь к процессу, более похожему на аукцион. Речь идёт об оценке увеличивающих цепей. Продемонстрируем аукцион на примере сети, представленной на рис. 26-1, где рядом с дугами показаны их цены (здесь цена, увы, только отчасти соотносится с длиной дуги).

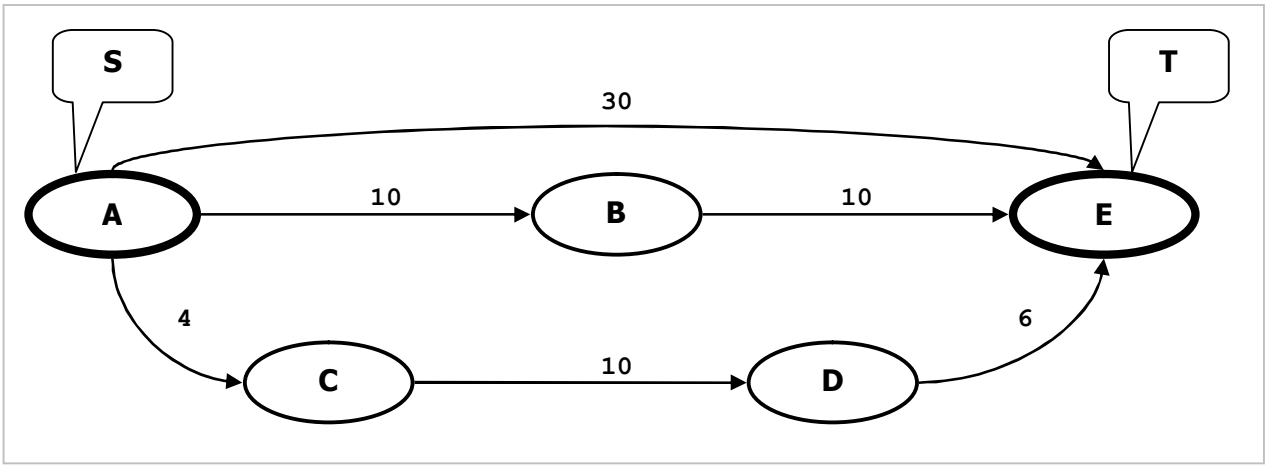


Рис. 26-1 — Сеть, где дугами назначены цены

Обратите внимание на следующее: при поиске *максимального* потока без учёта цены первой была бы выбрана цепь *A-E*, затем *A-B-E* и далее *A-C-D-E*. Но теперь всё иначе: мы вынуждены платить за цепочки, а первая из них отнюдь не самая дешёвая.

В ходе покупки будем постепенно поднимать цену (аукцион!), отмечая по ходу потенциально купленные участки сети так называемыми *вершинными числами*, — о том, как они формируются и применяются, сказано ниже.

На рис. 26-2 показано исходное состояние вершинных чисел перед началом аукциона (для этих чисел отведено поле **`TNode.mDist`**). Исток сети — это уже приобретённая вершина, и потому окрашен серым.

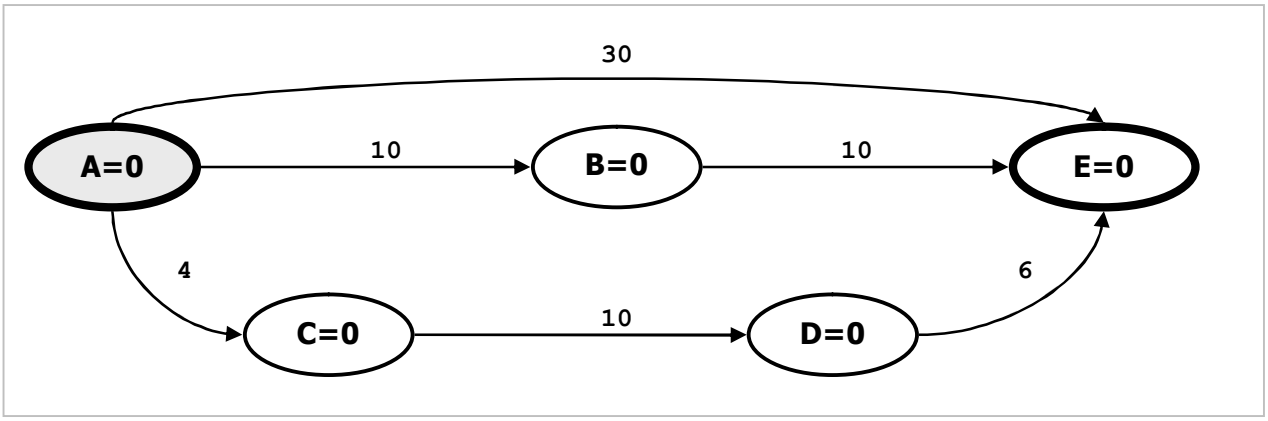


Рис. 26-2 — Исходные значения вершинных чисел

Теперь постепенно набавляем цену: **1**, **2**, **3**... пиастра, всякий раз наращивая вершинные числа *только в белых* вершинах. После каждой прибавки проверяем разность вершинных чисел соседних *разноцветных* вершин: когда эта разность *сровняется* с ценой соединяющей их дуги, то такая дуга и соответствующая ей концевая вершина считаются *приобретёнными*. Купленная вершина окрашивается серым, и потому её вершинное число далее *не наращивается*. Процесс завершается после приобретения стока сети T, и тогда вершинное число в стоке укажет стоимость всей приобретённой цепи (или нескольких равноценных). Важно учесть, что на торг выставляют только *прямые* дуги с *ненасыщенным* потоком (дуги класса I), — *нейтральные* дуги *не рассматриваются*. Если посредством дуг класса I не удаётся купить увеличивающую цепь, процесс завершается с отрицательным результатом. На рис. 26-3 показано состояние вершинных чисел в момент приобретения за **4** монеты дуги *A-C*.

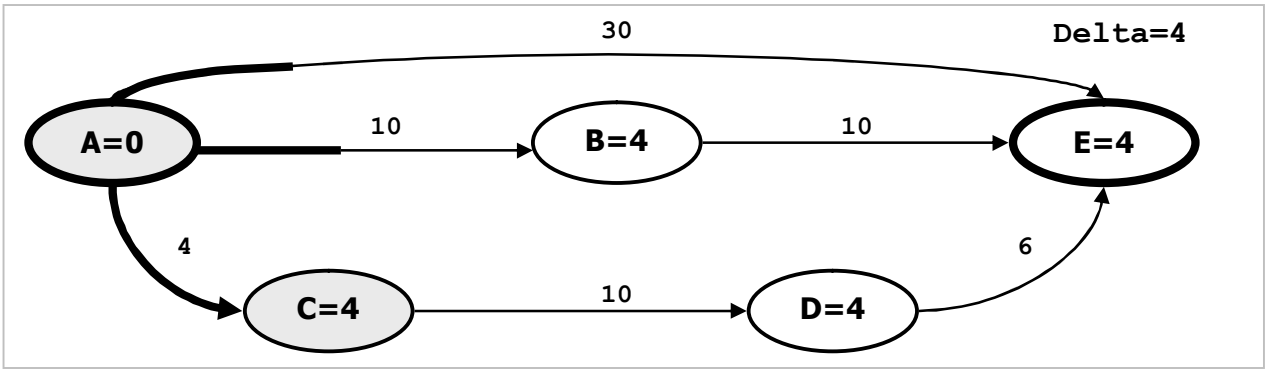


Рис. 26-3 — Участки сети, купленные за 4 монеты: приобретена дуга A-C

Понятно, что осторожное набавление цены с единичным шагом — не самый быстрый путь к цели. Ведь по разности чисел в соседних разноцветных вершинах и цене соединяющей их дуги можно сразу определить минимальную прибавку, необходимую для покупки этой дуги. Перебрав все дуги, соединяющие серые и белые вершины, и определив минимальную из минимальных прибавок, можно сразу предложить нужную сумму и купить одну из этих дуг. Формула для вычисления прибавки очень проста:

```
Delta = Cost - (G - W)
```

где:

- **`Delta`** — прибавка, необходимая для приобретения дуги;
- **`Cost`** — цена приобретаемой дуги;
- **`G`**, **`W`** — вершинные числа соответственно в серой и белой вершинах.

На последующих рисунках показаны минимальные прибавки и приобретённые участки сети (здесь все приобретаемые дуги — увеличивающие).

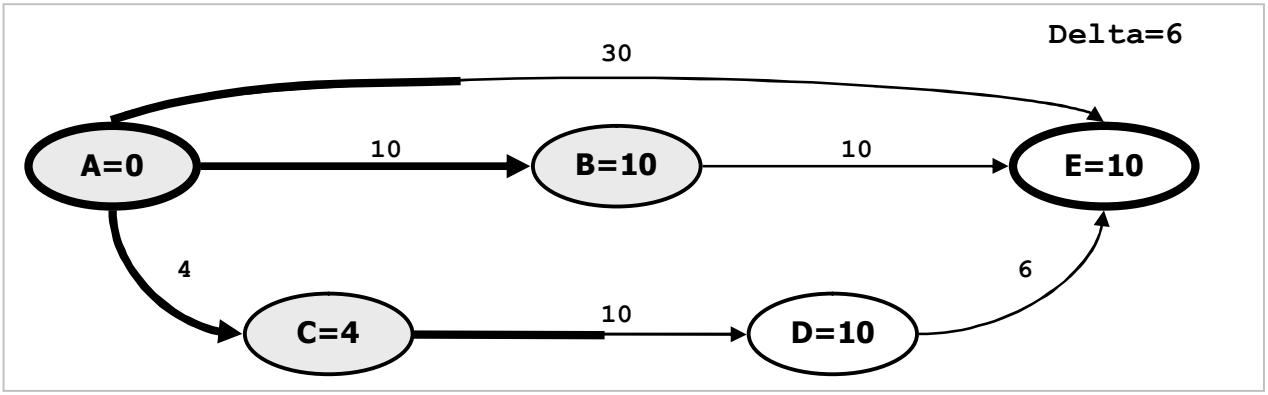


Рис. 26-4 — Добавлено 6 монет (всего 10), приобретена дуга A-B

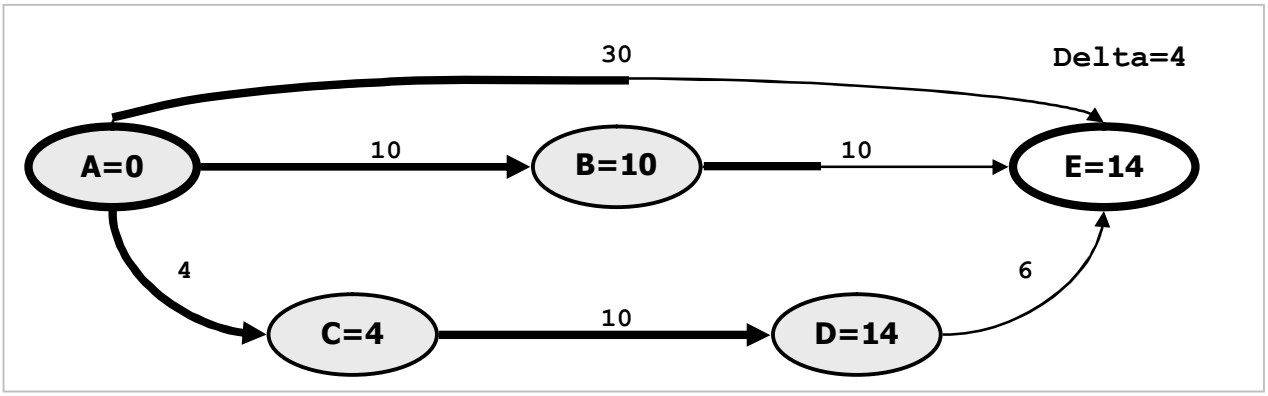


Рис. 26-5 — Добавлено 4 монеты (всего 14), приобретена дуга C-D

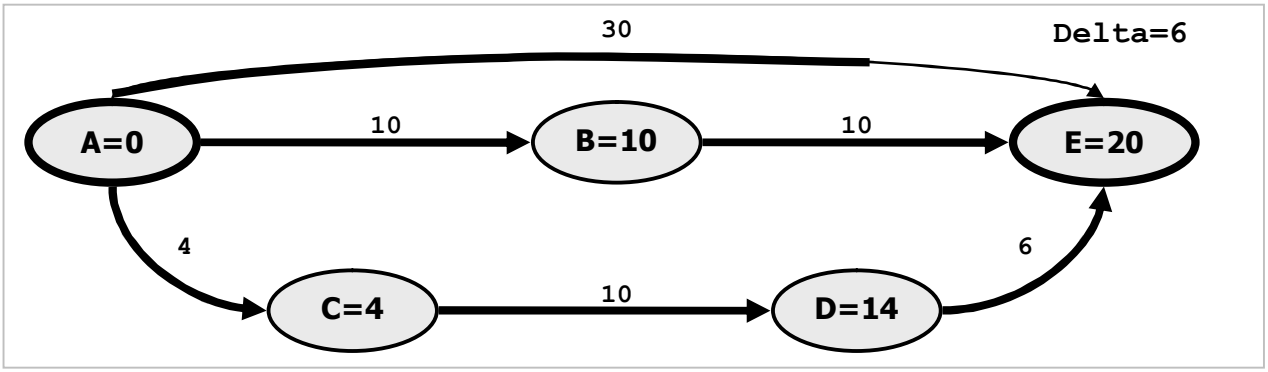


Рис. 26-6 — Добавлено 6 монет (всего 20), приобретены дуги B-E и D-E

Таким образом, после окраски стока у всех приобретённых в текущем торге дуг разности вершинных чисел сровняются с ценами этих дуг, что и означает их приобретение. Этим признаком воспользуемся позже в процедуре поиска максимального потока вдоль купленных цепей. Напомню, что вершинное число в стоке покажет цену единицы потока вдоль купленной цепи.

В нашем примере приобретено две цепи с ценой пересылки 20 монет за единицу потока. Пустив максимальный поток вдоль этих цепей, мы вновь вернёмся на аукцион, но повторно приобрести эти цепи уже не сможем, поскольку они «закроются» насыщенными дугами. Торг пойдёт по иной траектории, и нам придётся приобрести более дорогую дугу *A-E* по цене 30 монет за единицу потока. Таким образом, каждая следующая покупка будет обходиться нам всё дороже и дороже, пока не будут приобретены и насыщены цепи, пропускающие требуемый поток.

26.4. Метод вычисления стоимости потока

Теперь, после обсуждения основных идей, обратимся к листингу, где представлены детали вычисления стоимости потока (напомню, что *цена дуги* содержится в поле **`TLink.mValue`**).

Листинг 26-1 — Вычисление минимальной стоимости заданного потока

```
function TGraph.CalcMinCostFlow(aSource, aDest: TNode; // источник и сток
                                aFlow: integer         // величина потока
                                ): integer;            // стоимость потока

var  Gray: TSet;       // множество серых (купленных) вершин
     Que: TBuffer;  // очередь вершин используется в CalcDeltaFlow
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка данных потока и вершинных чисел,
  // вызывается единожды в начале метода
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure ClearFlowData;
  var Node: TNode;
      Link: TLink;      // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
```

```
      Node.mDist:= 0;       // очищаем вершинные числа
      // Перебор исходящих линков вершины:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mFlow:= 0;          // текущий поток = 0
        Link.mColor:= CWhite;    // цвет дуги белый
        Link:= Node.OutLinkNext;
      end; // while
      Node:= NodeNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета (mColor) и окраска вершины-истока
  // Вызывается в начале поиска очередной увеличивающей цепочки
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure ClearColors;
  var Node: TNode;       // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mColor:= CWhite;
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node:= NodeNext;
    end;
    // Красим вершину-исток и помещаем в множество серых вершин:
    aSource.mColor:= CGray;
    Gray.Clear;
    Gray.Insert(aSource);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Попытка приобретения очережной прямой дуги
  // Исходные данные:
  // - текущее множество серых вершин Gray
  // - текущий поток в дугах
  // - текущее состояние вершинных чисел
  // Возвращает true в случае покупки дуги, изменяя вершинные числа
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  function Buying: boolean;
  var Node: TNode;       // текущая вершина
      Link: TLink;       // текущий линк
      Delta: integer;    // очередное приращение суммы
      MinDelta: integer;// минимальное приращение суммы
  begin
    MinDelta:= MaxInt;
    // Перебор серых вершин:
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      // Просмотр исходящих дуг:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor = CWhite then begin
          // Если поток не насыщен:
          if mFlow < mHigh then begin
            // Прицениваемся к увеличивающей дуге:
            // Link.mValue - цена увеличивающей дуги
            // mOwner.mDist - сумма оплаты цепочки вплоть до Node
            // mDest.mDist - частичная плата вплоть до mDest
            // Delta - то, что нужно доплатить за приобретение дуги Link
            Delta:= mValue - (mDest.mDist - mOwner.mDist);
            if Delta < MinDelta
              then MinDelta:= Delta;  // запоминаем минимальную доплату
          end; // if
```

```
        end;
        Link:= Node.OutLinkNext;
      end; // while
      Node:= Gray.GetNext as TNode; // Следующая вершина серого множества
    end; // while Assigned(Node)
    Result:= MinDelta < MaxInt;     // true, если дуга куплена
    if not Result or (MinDelta=0) then Exit;
    // Наращиваем mDist - вершинные числа белых вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor = CWhite then Inc(Node.mDist, MinDelta);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск потока, проходящего по уже купленным дугам
  // Исходные данные:
  // - текущее множество серых вершин
  // - текущий поток
  // - текущие вершинные числа
  // Расширяет множество серых вершин.
  // Возвращает допустимое приращение потока (или ноль)
  // и стоимость единицы потока
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  function CalcDeltaFlow(var aCost: integer): integer;
  var
      Node: TNode;      // текущая вершина
      Link: TLink;      // исходящие и входящие дуги
  begin
    Result:=0; aCost:=0;
    Que.Clear;  // очищаем очередь вершин
    // Ставим в очередь серые вершины
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      Que.Put(Node);
      Node:= Gray.GetNext as TNode;
    end;
    Que.Put(aSource);
    // Обработка вершин из очереди:
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Обработка исходящих дуг текущего узла:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor=CWhite then begin
          // если вершина не помечена
          // Помечаем только белые вершины,
          // достигаемые через купленые ненасыщенные дуги
          // Link.mValue -- стоимость дуги
          // Link.mDest.mDist - mOwner.mDist -- разность вершинных чисел
          if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
             and (mFlow < mHigh) then begin  // и дуга не насыщена
            mDirect:= +1;  // отмечаем прямую дугу
            // помечаем концевую вершину и ставим в очередь:
            mDest.mColor:= CGray;     // цвет концевой вершины серый
            mDest.mPred:= Node;       // предшествующая вершина
            mDest.mLink:= Link;       // линк на предшествующую вершину
            mDest.mFlow:= Minimum(Node.mFlow, mHigh - mFlow); // поток
            Que.Put(mDest);       // в очередь приёмник дуги
            Gray.Insert(mDest);   // пополняем множество окрашенных
          end;
        end; // if
```

```
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        with Link do if mOwner.mColor=CWhite then begin
          // Помечаем только белые вершины,
          // из которых поступает ненулевой поток
          if mFlow > 0 then begin  // поток в дуге можно уменьшить
            mDirect:= -1;  // отмечаем встречную дугу
            // помечаем вершину и ставим в очередь:
            with Link.mOwner do begin
              mColor:= CGray;     // цвет исходной вершины серый
              mPred:= Node;       // предшествующая вершина
              mLink:= Link;       // линк на предшествующую вершину
              mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
            end;
            Que.Put(Link.mOwner);      // в очередь источник дуги
            Gray.Insert(Link.mOwner); // пополняем множество окрашенных
          end;
        end;
        Link:= Node.InLinkNext;
      end;  // while Assigned(Link)
      // Если помечен сток, то выход из цикла:
      if aDest.mColor <>  CWhite then Break;
    end;  // while Que.GetCount>0
    // Если сток помечен, то увеличивающая цепочка найдена
    // возвращаем допустимое увеличение потока и цену единицы потока
    if aDest.mColor <> CWhite then begin
      Result:= aDest.mFlow;
      aCost:= aDest.mDist;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Пересчёт потока вдоль предварительно найденной цепочки
  // aDelta - приращение потока
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure RecalcFlow(aDelta: integer);
  var Node: TNode;     // текущая вершина
  begin
    // Обратное движение от стока к истоку с корректировкой потока в дугах
    Node:= aDest;  // сток
    // Цикл, пока не достигнем истока:
    while Node <> aSource do begin
      // Node.mLink - линк из предшествующей вершины
      // Node.mLink.mDirect = +1 / -1
      with Node.mLink do Inc(mFlow, mDirect * aDelta); // поток
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Flow: integer;         // накопленный поток
  DeltaFlow: integer;    // приращение потока
  Cost1: integer;        // цена единицы потока
  OK: boolean;           // признак приобретения дуги

begin { TGraph.CalcMinCostFlow }

  Result:= 0;        // накопленная стоимость потока
  Flow:= 0;          // накопленный поток
  Gray:= CreateSet; // множество окрашенных (купленных) вершин
```

```
  Que:= TBuffer.Create;  // создаём очередь вершин
  ClearFlowData;     // очистка потока в дугах (mFlow=MaxInt)
  ClearColors;       // очистка цвета и начальная установка серого множества
  OK:= false;        // признак приобретения дуги

  // Пока поток не достиг заданного значения:
  while Flow < aFlow do begin
    // Пытаемся "купить" увеличивающую цепочку (дугу)
    // и формируем вершинные числа.
    // Если куплена хотя бы одна дуга, пытаемся провести поток:
    repeat
      // Пытаемся провести поток и определить цену единицы потока
      DeltaFlow:= CalcDeltaFlow(Cost1);
      if DeltaFlow > 0  then begin
        // Поток возможен, ограничиваем его приращение:
        if DeltaFlow > aFlow - Flow then DeltaFlow:= aFlow - Flow;
        // Распределяем поток по дугам и и определяем цену единицы потока:
        RecalcFlow(DeltaFlow);         // пересчёт потока в дугах
        Inc(Flow, DeltaFlow);          // суммарный поток
        Inc(Result, Cost1*DeltaFlow); // суммарная стоимость потока
        ClearColors;                   // очистка цвета и серого множества
      end else begin
        // Если поток не проведен, пытаемся купить хотя бы одну дугу
        // (функция Buying модифицирует вершинные числа)
        OK:= Buying;
        // Если цепочку купить нельзя, то заданный поток не существует
        if not OK then Break;
      end;
      // пока поток не достигнет нужного, либо не найдено приращение
    until Flow = aFlow;
    // Если цепочку купить нельзя, то заданный поток не существует
    if not OK then begin
      Result:=-1;  // стоимость = -1
      Break;
    end;
  end; // while Flow < aFlow
  // Очистка памяти:
  Gray.Free; // серое множество
  Que.Free;  // рабочая очередь
end;
```

Здесь функция **CalcDeltaFlow**, вычисляющая допустимое приращение потока вдоль очередной цепи, почти совпадает с аналогичной функцией в методе поиска *максимального* потока. Единственное отличие состоит в дополнительной проверке, куплена ли дуга:

```
          if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
             and (mFlow < mHigh) then begin        // и дуга не насыщена
```

Функция покупки очередной дуги **CalcCost1** в целом соответствует обсуждённому выше алгоритму.

26.5. Тестирование

Программа для проверки метода вычисления минимальной стоимости потока представлена ниже.

Листинг 26-2 — Минимальная цена потока

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CFile = 'Test.txt';

var Gr : TGraph;
    S, T : TNode;
    Flow: integer;
    Cost: integer;
begin
    Gr:= TGraphChars.Load(CFile);
    (Gr as TGraphChars).LoadFlowData(CFile);
    S:= (Gr as TGraphChars).GetNode('A');
    T:= (Gr as TGraphChars).GetNode('G');
    Flow:= Gr.CalcMaxFlow0(S, T);
    Writeln('MaxFlow= ', Flow);
    repeat
      Write('Flow = '); Readln(Flow);
      Cost:= Gr.CalcMinCostFlow(S, T, Flow);
      Writeln('Cost= ', Cost);
      Gr.ExpoLinksData;
    until Flow=0;
    Gr.Free;
end.
```

Для тестирования взят граф, изображённый на рис. 26-7, данные для его ввода представлены ниже (стоимость дуги задана полем **mValue**).

```
Test.txt - cost of flow (Kristofides, 342)
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
7 - количество вершин
A B C D E F G
A -> B=7 D=13 F=28
B -> C=25 D=4 E=10
C -> G=5
D -> C=6 F=5
E -> G=12
F -> E=3 G=7
G ->

FLOW:
AB= 0 16
AD= 0 11
AF= 0 13
BC= 0 17
BD= 0 18
BE= 0 16
CG= 0 22
DC= 0 12
DF= 0 19
EG= 0 16
FE= 0 10
FG= 0  5
```

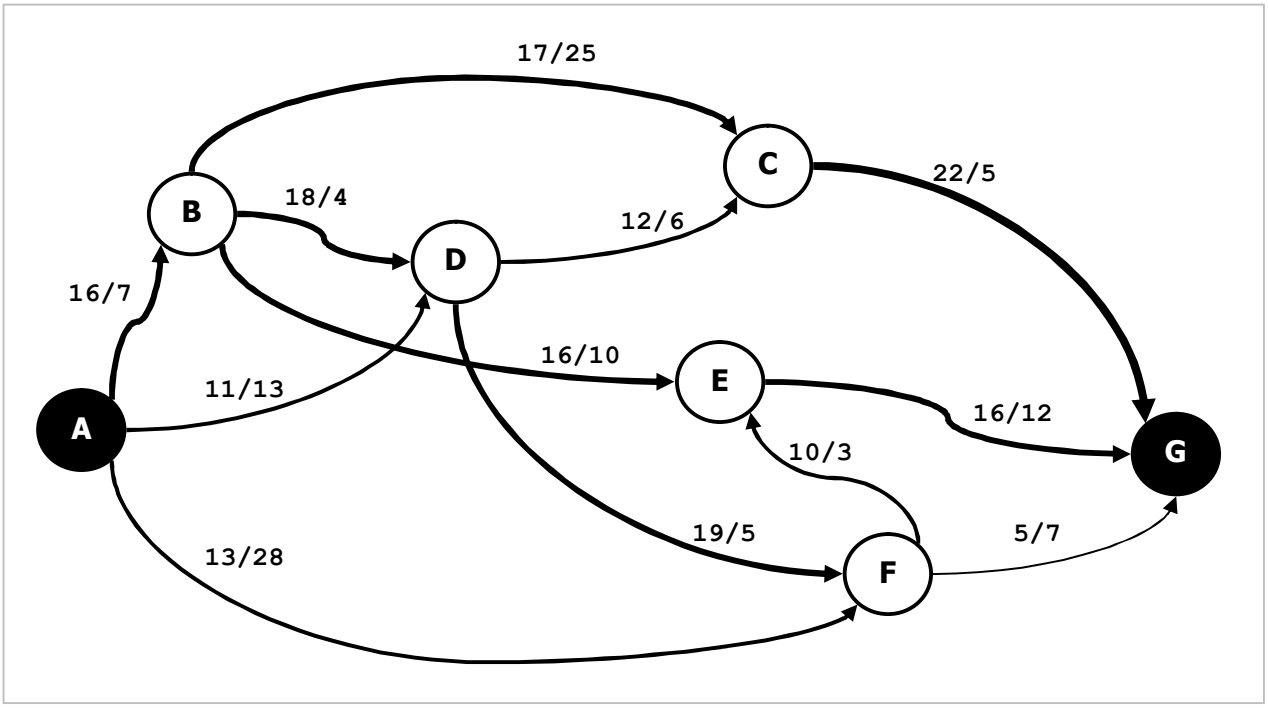


Рис. 26-7 — Граф для испытания алгоритма поиска минимальной стоимости потока (числитель — пропускная способность, знаменатель — цена дуги)

На следующих рисунках показаны найденные цепочки, их цены и пропускные способности, а также вершинные числа и распределение потока в дугах. Каждый рисунок соответствует покупке очередной цепи, увеличивающей поток.

Обратите внимание на изменение потока в обратных дугах. Так, например, пущенный на первых итерациях по дуге *D-F* поток в **5** единиц (рис. 26-11), позже перераспределяется так, что полностью уходит по другим маршрутам, и дуга *D-F* остаётся в итоге «сухой» (рис. 26-12).

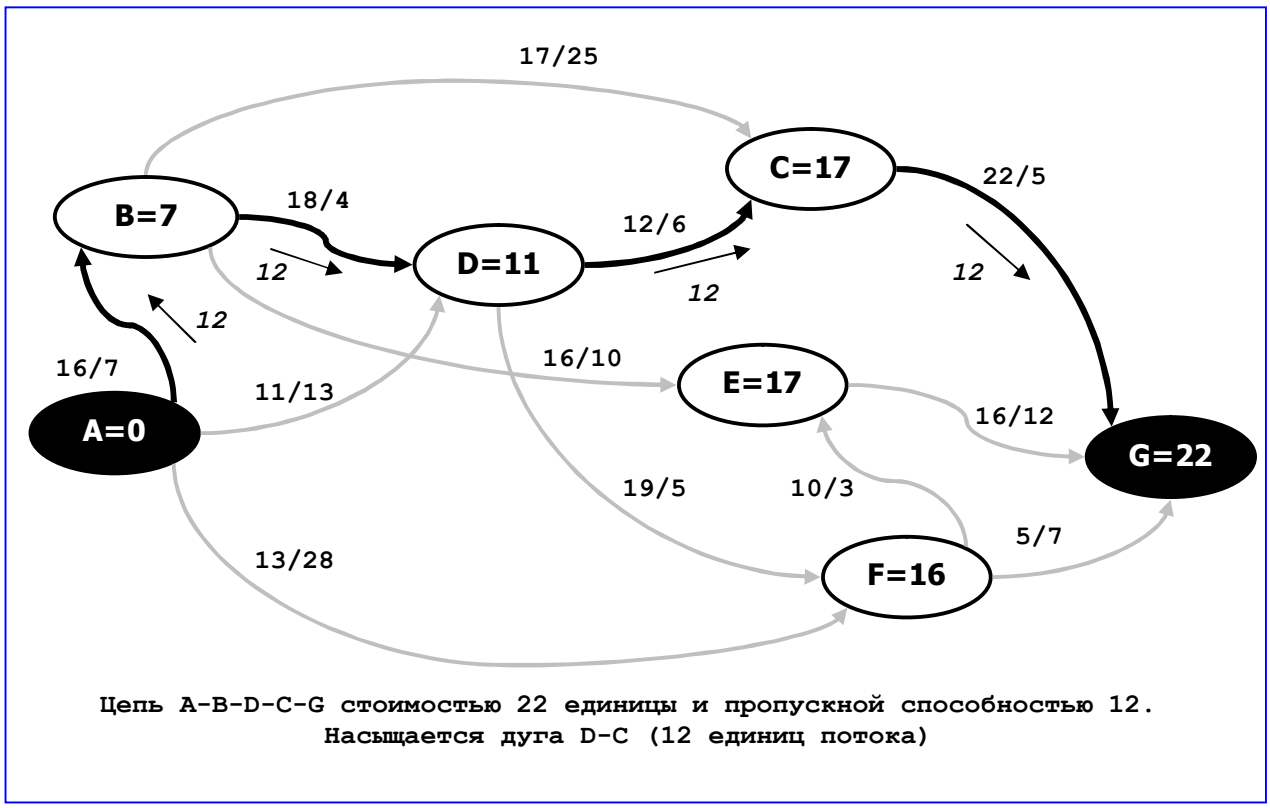


Рис. 26-8 — Первая цепочка стоимостью 22 единицы

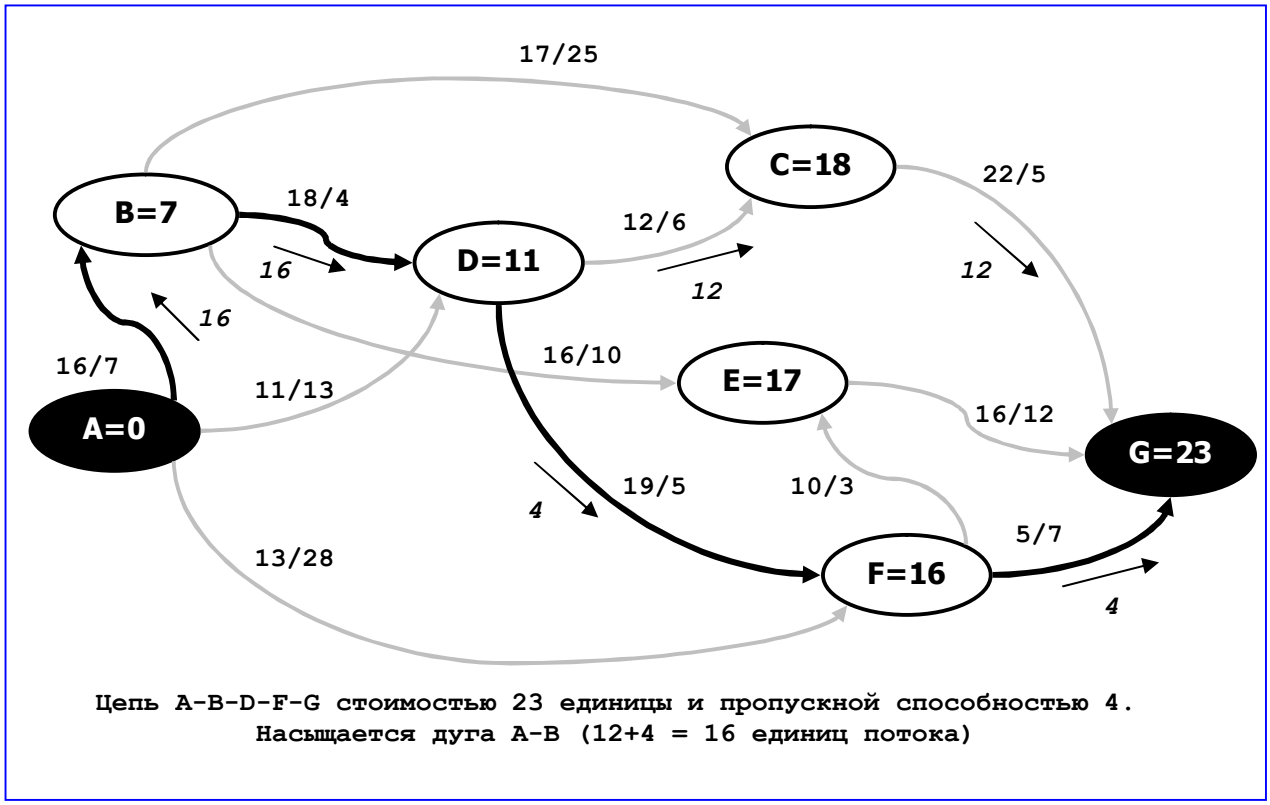


Рис. 26-9 — Вторая цепочка стоимостью 23 единицы

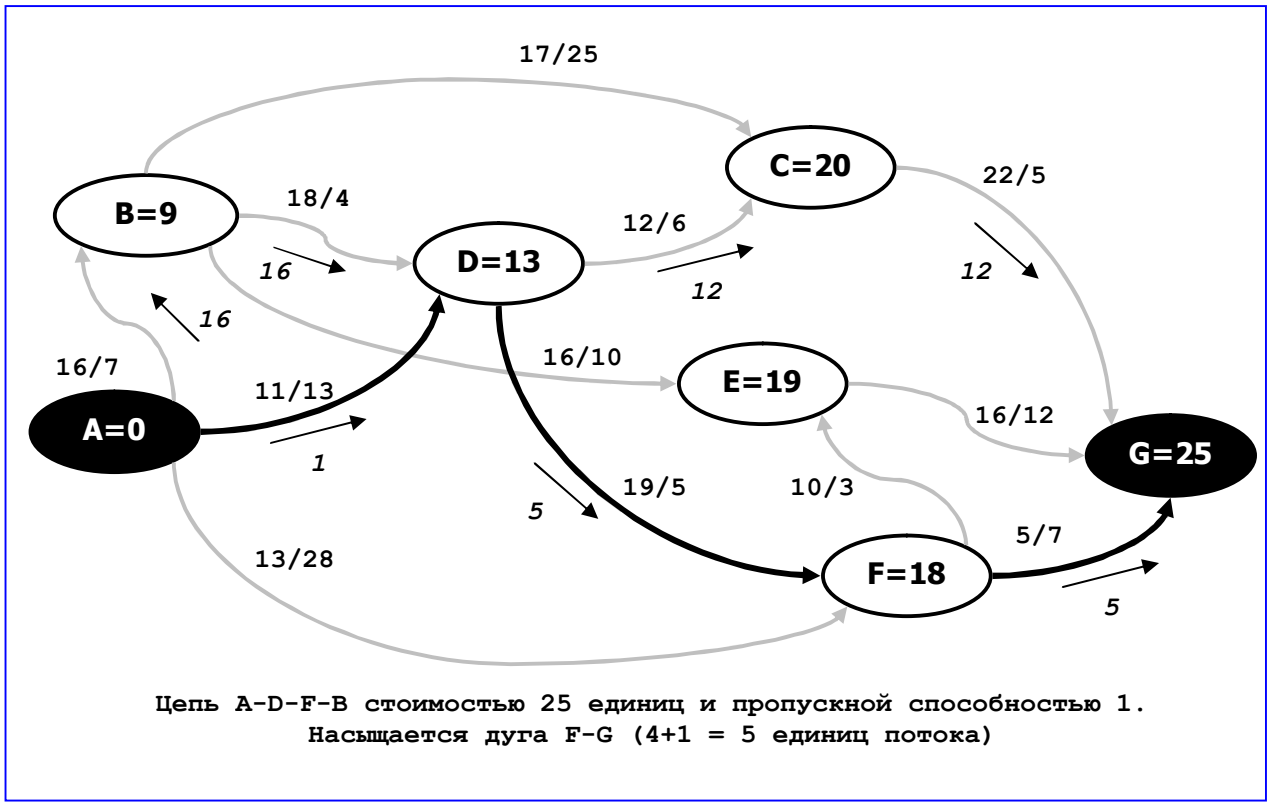
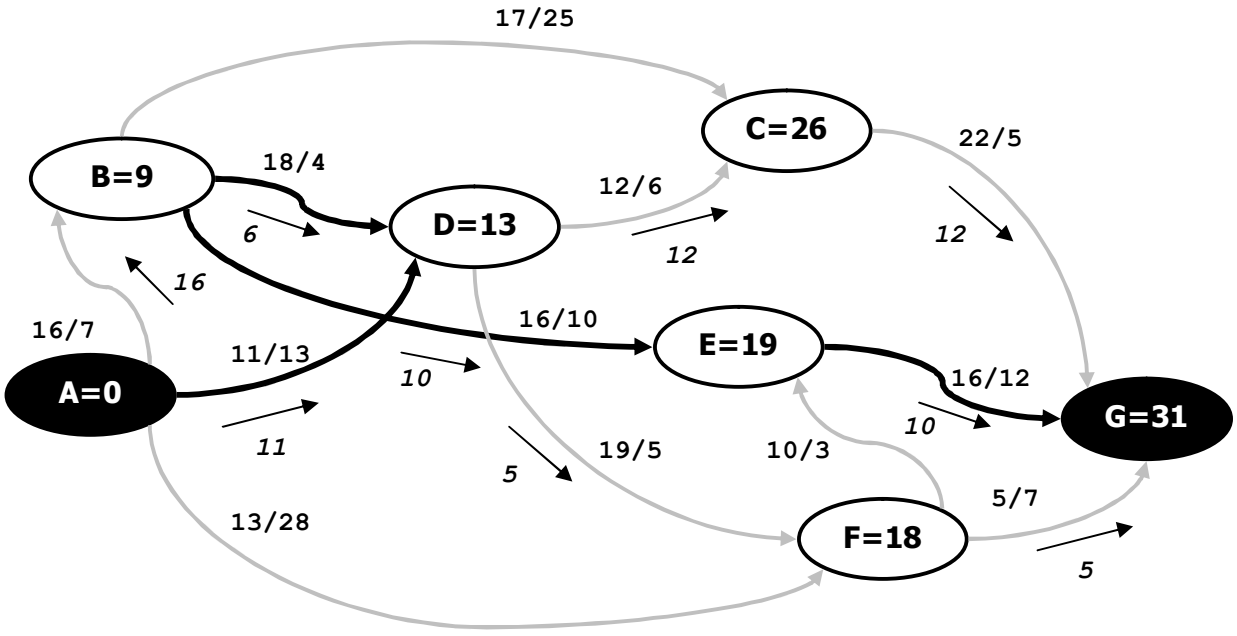
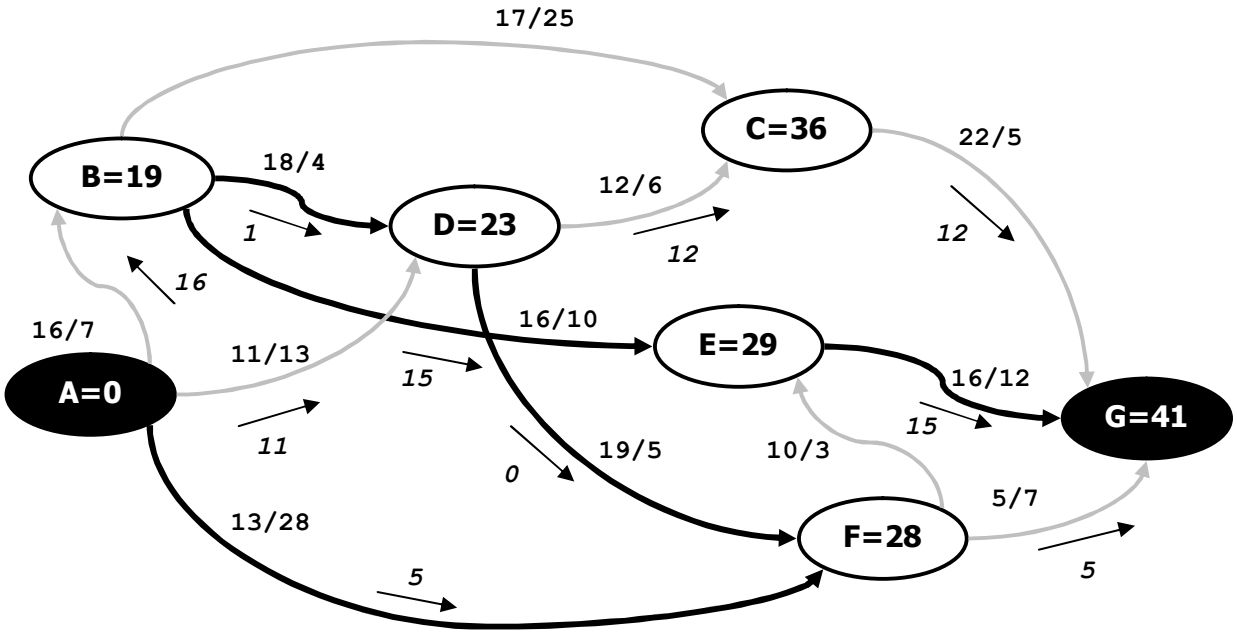


Рис. 26-10 — Третья цепочка стоимостью 25 единиц



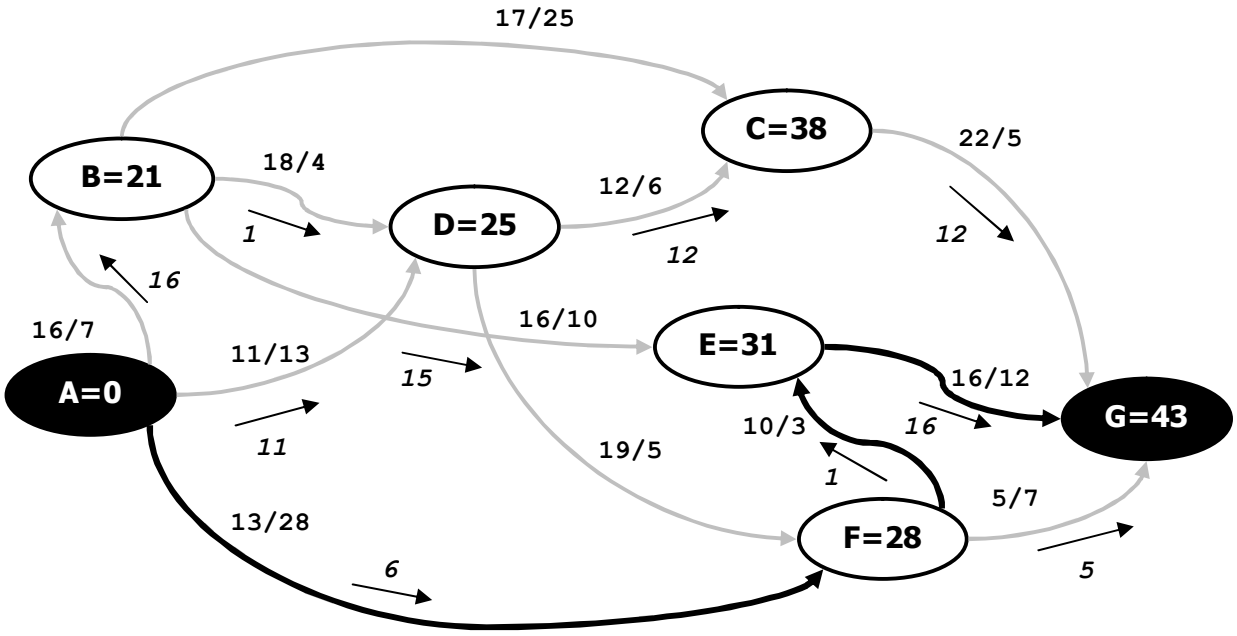
Цепь A-D-B-E-G стоимостью 31 единица и пропускной способностью 10.
Насыщается дуга A-D (1+10 = 11 единиц потока).
Поток в обратной дуге B-D уменьшается с 16 до 6 единиц.

Рис. 26-11 — Четвёртая цепочка стоимостью 31 единица



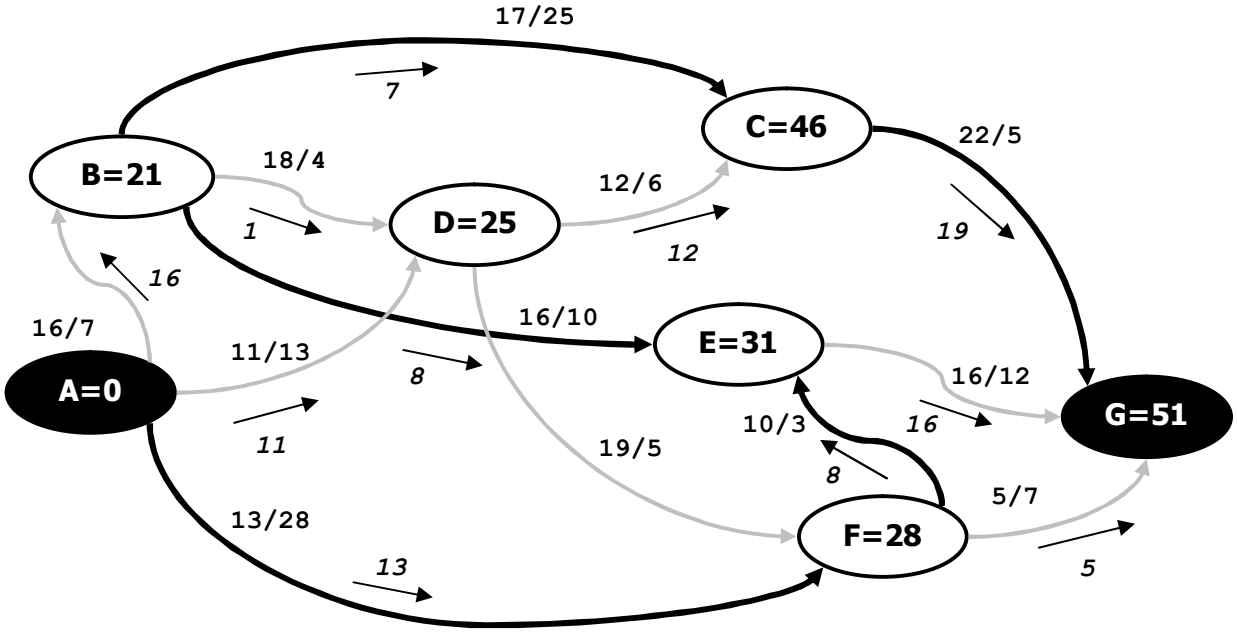
Цепь A-F-D-B-E-G стоимостью 41 единица и пропускной способностью 5.
Исчезает поток в обратной дуге D-F (5-5 = 0 единиц потока).
Поток в обратной дуге B-D уменьшился с 6 до 1 единицы.

Рис. 26-12 — Пятая цепочка стоимостью 41 единица



Цепь A-F-E-G стоимостью 43 единицы и пропускной способностью 1.
Насыщается дуга E-G (15+1 = 16 единиц потока).

Рис. 26-13 — Шестая цепочка стоимостью 43 единицы



Цепь A-F-E-B-C-G стоимостью 51 единица и пропускной способностью 7.
Насыщается дуга A-F (6+7 = 13 единиц потока).
Поток в обратной дуге B-E уменьшился с 15 до 8 единиц.

Рис. 26-14 — Седьмая цепочка стоимостью 51 единица

Приведенные выше рисунки с этапами и результатами расчёта убеждают в справедливости следующих утверждений:

- Сумма потоков, исходящих из истока S, равна сумме потоков, поглощаемых стоком T.
- Сумма потоков, входящих в промежуточную вершину, и исходящих из неё, равна нулю.
- Разность вершинных чисел смежных вершин вдоль вновь приобретённой цепи равна цене инцидентных им дуг.
- Вершинное число в стоке T после покупки цепи равно стоимости вновь приобретённой цепи.
- Максимальный поток вдоль очередной цепи либо насыщает прямую дугу, либо истощает обратную. И те, и другие дуги в ходе следующей итерации будут трактоваться как нейтральные, и потому не влияют на дальнейшее распределение потока.

Итоги вычислений сведены в табл. 26-1.

Табл. 26-1 — Сводная таблица с результатами расчёта минимальной стоимости потоков

Цена единицы потока и цепь	Величина потока	Стоимость	Цена единицы потока и цепь	Величина потока	Стоимость
22 A-B-D-C-G	1	22	31 A-D-B-E-G	21	505
	2	44		22	536
	3	66		23	567
	4	88		24	598
	5	110		25	629
	6	132		26	660
	7	154		27	691
	8	176	41 A-F-D-B-E-G	28	732
	9	198		29	773
	10	220		30	814
	11	242		31	855
	12	264		32	896
23 A-B-D-F-G	13	287	43 A-F-E-G	33	939
	14	310	51 A-F-E-B-C-G	34	990
	15	333		35	1041
	16	356		36	1092
25 A-D-F-G	17	381		37	1143
31 A-D-B-E-G	18	412		38	1194
	19	443		39	1245
	20	474		40	1296

26.6. Итоги

26.6.1. Задача поиска минимальной стоимости потока состоит в том, чтобы при заданных пропускных способностях дуг и их стоимостях наиболее рационально распределить поток по дугам сети, и попутно определить стоимость этого потока.

26.6.2. Задача решается «покупкой» увеличивающих цепей, начиная с самых «дешёвых». Через приобретённые цепочки проводят допускаемый ими поток, после чего некоторые прямые дуги сети насыщаются либо истощаются обратные. Те и другие не рассматриваются в ходе последующих «торгов».

26.7. Задания

26.8. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 339, Стр. 351
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 104
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 27
Паросочетание в двудольном графе

Формирование пар, или бракосочетание — задача, решаемая природой с приходом каждой весны. Вот другие примеры: оптимальная расстановка рабочих по станкам, формирование спортивных пар.

27.1. Двудольные графы

Задачи этого рода моделируют *двудольными* графами. *Двудольным* называют граф, все вершины которого разбиты на два подмножества — назовём их условно левым и правым. При этом все вершины левого подмножества связаны рёбрами или дугами *только* с вершинами правого подмножества, и наоборот. Примеры: покупатели и товары, рабочие и станки. Рассмотрим ещё пример.

Некий торговец недвижимостью хочет продать с наибольшей выгодой ряд домов, для чего он подыскал несколько покупателей и заранее ознакомил их с товаром. Каждый покупатель оговорил приемлемую для себя цену нескольким приглянувшимся ему домам, намереваясь, в конечном счете, купить один из них. Таким образом, каждый покупатель заинтересовался несколькими домами, и каждый дом приглянулся нескольким покупателям, назначившим за него каждый свою цену. Теперь дело за продавцом: он должен распределить дома между покупателями так, чтобы сумма всех сделок оказалась максимальной. Такова задача поиска паросочетания максимальной стоимости на двудольном графе, где левую долю составляют покупатели, а правую — товары. Здесь мы стремимся *максимизировать* сумму, в других задачах требуется *минимизировать* что-то (расходы, убытки), — оба варианта задачи решаются одним алгоритмом, поскольку взаимно преобразуются друг в друга.

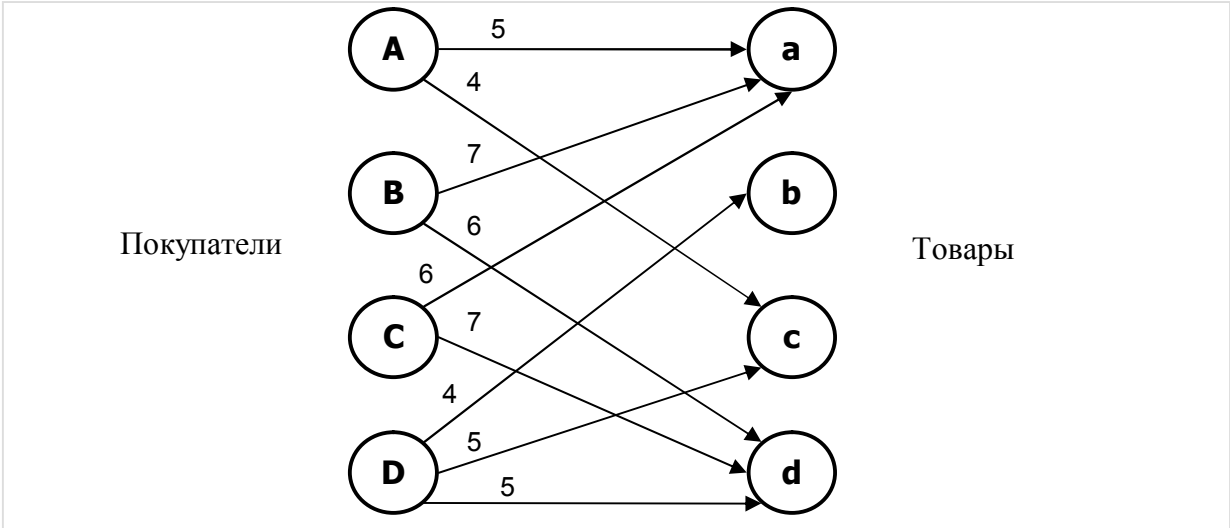


Рис. 27-1 — Покупатели (слева) и товары (справа)

Двудольный граф на рис. 27-1 отражает ситуацию с покупателями (A, B, C, D) и домами (a, b, c, d). Числа у стрелок означают цену, которую готов уплатить

покупатель за тот или иной дом. Здесь домом *a* заинтересовались три покупателя, а домом *b* лишь один. Количество покупателей и товаров может не совпадать, и тогда либо кто-то останется без дома, либо часть домов окажется не проданной.

Тем же рисунком можно описать другой случай. Пусть вершинам левой доли соответствуют источники товара (фабрики или оптовые базы), а вершинам правой доли — его потребители (сеть розничных магазинов). Стрелки показывают возможные пути доставки и стоимость доставки товара. Пусть каждый источник товара способен обеспечить лишь один магазин. Тогда хозяину розничной сети надо минимизировать расходы на доставку: найти *наибольшее* паросочетание *минимальной* стоимости.

Итак, решаемая задача формулируется следующим образом. Дан двудольный граф, вершины которого явно разбиты на два подмножества: левое и правое. Левое подмножество соединяется с правым взвешенными дугами или рёбрами. Необходимо найти наибольшее множество дуг (либо соответствующие им пары вершин) суммарная стоимость которых окажется минимальной.

27.2. Термины, определения и очевидные закономерности

Обсуждение путей решения задач о паросочетаниях предварим знакомством с терминами, принятыми в этой сфере.

Прежде всего, что такое *паросочетание* ? Это подмножество попарно несмежных рёбер графа, то есть рёбер, не имеющих общих вершин. Отметим, что *пустое* множество рёбер тоже является паросочетанием.

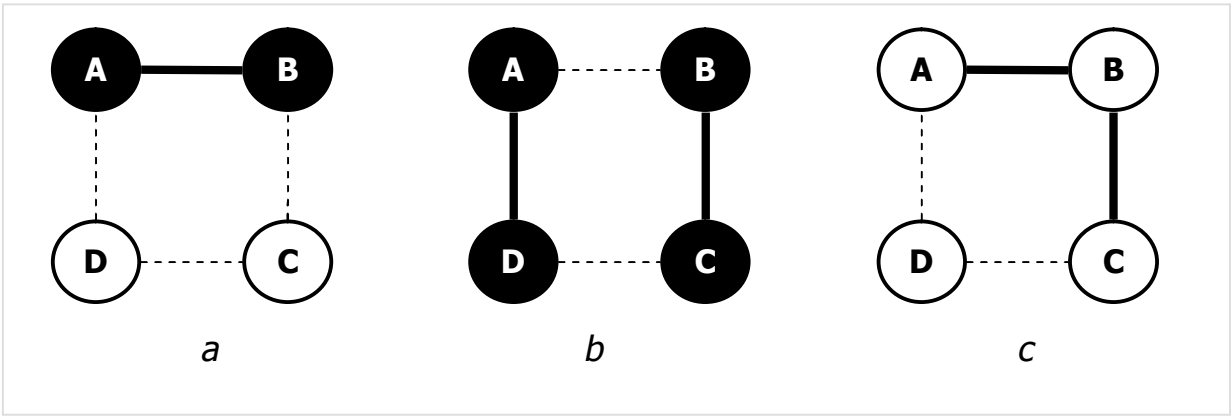


Рис. 27-2 — Паросочетания (a) и (b)

Обратимся к рис. 27-2. Условимся далее рёбра, не входящие в паросочетание, рисовать тонким пунктиром и условно считать *белыми*, а рёбра, входящие в него, выделять сплошной жирной линией и условно считать *чёрными*. Таким образом, на рисунке (*a*) мы видим паросочетание из одного ребра *A-B*, а на рисунке (*b*) — паросочетание из двух рёбер: *A-D* и *B-C*. На рисунке (*c*) выделено жирным два ребра, но они *не составляют* паросочетания, поскольку имеют одну общую вершину *B*.

Вершины, не инцидентные рёбрам паросочетания, называют *свободными* или *экспонированными*. На рисунке (*a*) это вершины *D* и *C*, мы будем красить их *белым*. Вершины, соединённые ребром паросочетания, окрасим чёрным.

Из приведенных выше определений следуют очевидные выводы:

- у чёрного ребра (входящего в паросочетание) обе смежные вершины чёрные;
- у белого ребра хотя бы одна из двух инцидентных вершин белая;
- у чёрной вершины хотя бы одно смежное ребро чёрное;
- у белой вершины все смежные рёбра белые.

Паросочетание, которое невозможно расширить добавлением рёбер, не удалив из него уже имеющееся, называется *максимальным*, а паросочетание, содержащее наибольшее количество рёбер, — *наибольшим*. На рис. 27-3 показаны максимальное паросочетание (a), и два наибольших (b, c). И тех, и других, в графе может быть несколько. Отметим, что к паре *A-C* невозможно присоединить другие рёбра, и потому такое паросочетание является максимальным, хотя и не наибольшим. *Наибольшее* паросочетание является одновременно и *максимальным*, но не наоборот.

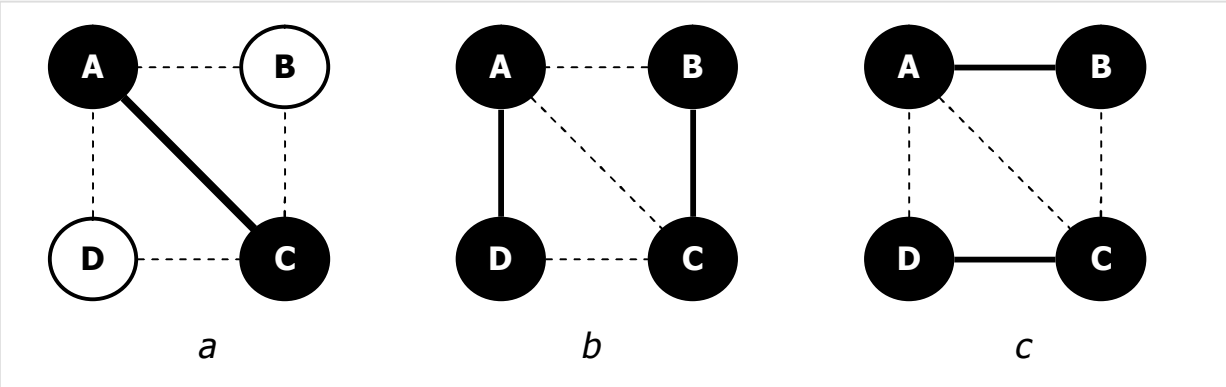


Рис. 27-3 — Максимальное (a) и два наибольших (b, c) паросочетания

Совершенным называют паросочетание, в которое вовлечены все вершины, — такой граф должен содержать чётное количество вершин. *Совершенное* паросочетание одновременно является и *наибольшим*.

Оптимальным назовём такое *наибольшее* паросочетание на взвешенных рёбрах, у которого сумма весов рёбер либо *минимальна*, либо *максимальна* (зависит от постановки задачи). Таким образом, в упомянутых выше задачах ищется одно из оптимальных паросочетаний: минимального либо максимального веса.

27.3. Увеличивающие цепи и деревья

Подход, принятый в алгоритмах поиска *наибольших* паросочетаний состоит в постепенном добавлении рёбер к текущему паросочетанию, начиная с пустого. Для

описания процесса нужны ещё два термина: чередующаяся (*альтернирующая*) цепь, и увеличивающая (*аугментальная*) цепь.

Альтернирующая цепь состоит из попарно смежных рёбер, *чередующихся* так, что за ребром не входящим в паросочетание следует ребро, входящее в него, и т.д.; примеры представлены на рис. 27-4. *Альтернирующие* цепи в свою очередь могут оказаться:

- a) увеличивающими (*аугментальными*);
- b) уменьшающими;
- c) нейтральными (*венгерскими*).

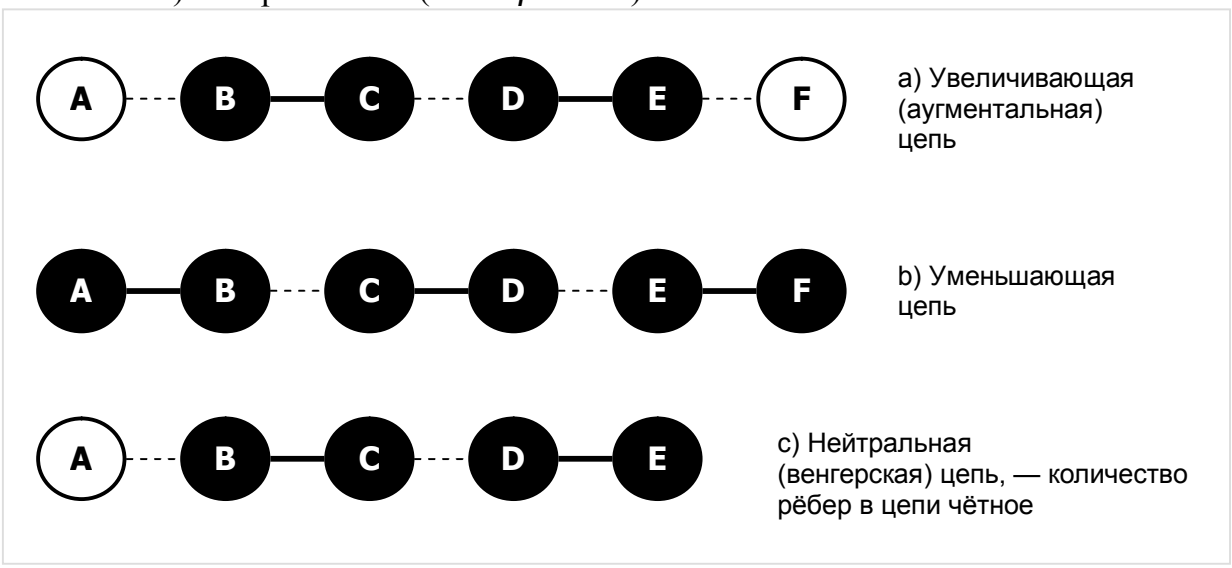


Рис. 27-4 — Альтернирующие (чередующиеся) цепи

Аугментальная цепь характерна тем, что два крайних ребра и две крайние вершины в ней — белые (свободные). Цепь хороша тем, что если рёбра такой цепи инвертировать — перекрасить в противоположные цвета (белые в чёрные, а чёрные в белые), — то в паросочетании станет на одно чёрное ребро больше, а цепь тем самым превратится в уменьшающую. Этим качеством не обладает нейтральная (венгерская) цепь. Отсюда следует, что для постепенного расширения паросочетания надо отыскивать *аугментальные* цепи.

27.4. Поиск увеличивающих (аугментальных) цепей

Поиск исходит из того, что *увеличивающая* цепь всегда начинается с белой (экспонированной) вершины и замыкается такой же. Стало быть, искать увеличивающую цепь надо начинать с *белой* вершины; поиск может завершиться:

- a) успешно — обнаружением белой вершины, либо
- b) неуспешно — после посещения всех доступных вершин графа.

Поскольку из белой вершины в общем случае можно следовать по нескольким направлениям, в ходе поиска будет порождаться дерево с корнем в этой вершине.

Дерево можно формировать обходом либо в глубину, либо в ширину. Рассмотрим обход в ширину на примере графа, представленного на рис. 27-5.

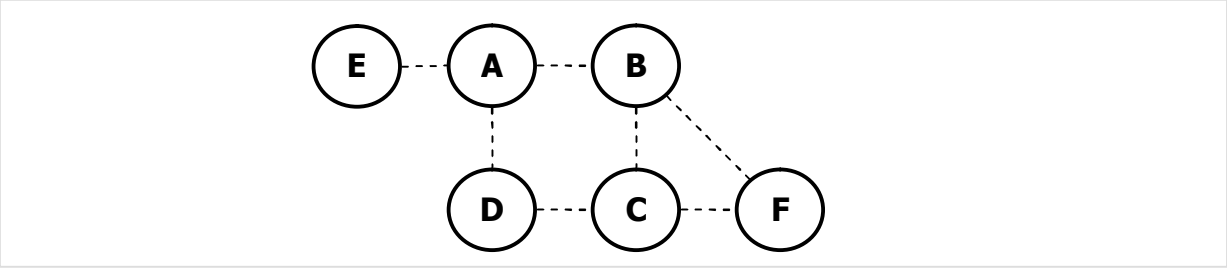


Рис. 27-5 — Исходный граф для поиска паросочетания

Станем обрабатывать вершины в алфавитном порядке, и начнём строить дерево из вершины *A*. Отсюда мы сразу попадаем в белую вершину *B*, и на этом построение дерева завершится, поскольку найдена увеличивающая цепь *A-B*. Инвертировав её, мы добавим в паросочетание первое ребро (рис. 27-6).

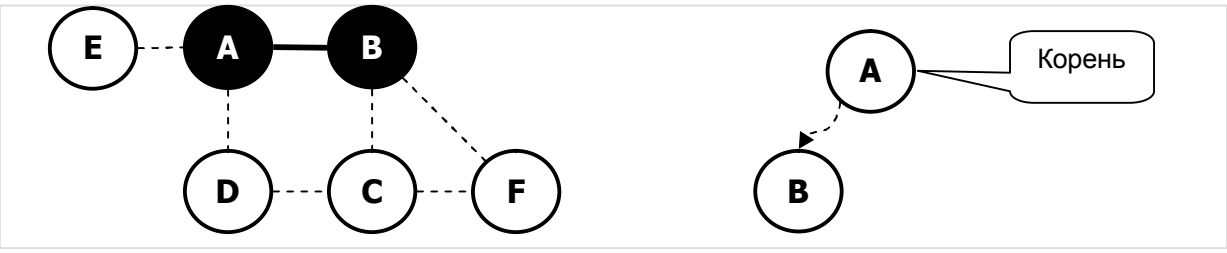


Рис. 27-6 — Построение цепи из вершины A, найдено ребро A-B

На следующей итерации дерево строится из белого корня *C*, и точно так же к множеству добавляется ребро *C-D* (рис. 27-7), а в графе остаются лишь две белые вершины.

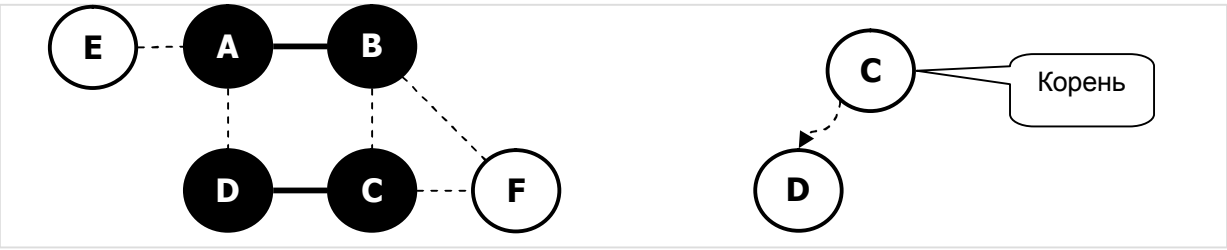


Рис. 27-7 — Построение цепи из вершины C, найдено ребро C-D

Теперь строим дерево из белого корня *E*, и здесь мы сразу попадём в чёрную вершину *A*. Казалось бы, отсюда можно следовать далее в вершины *B* и *D*. Но дело в том, что ребро *A-D* — белое (так же, как и предшествующее ему ребро *E-A*), и потому не даст чередующейся цепи. Поэтому из первой чёрной вершины *A* (её называют *внутренней*) можно проследовать далее только по чёрному ребру в чёрную вершину *B* (её называют *внешней*), соблюдая тем самым чередование рёбер. Из *внешней* чёрной вершины *B* мы вправе следовать далее по нескольким белам рёбрам, и так попадаем в две вершины: чёрную *C* и белую *F*. Вершиной *F* постройка дерева завершается (рис. 27-8), поскольку очередная увеличивающая цепь найдена.

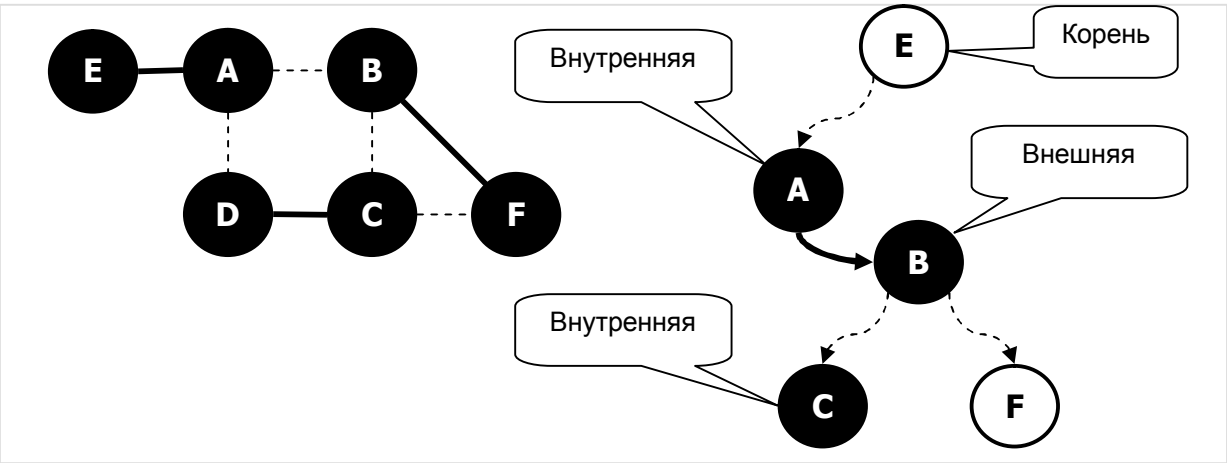


Рис. 27-8 — Построение цепи из вершины E, найдена увеличивающая цепь E-A=B-F

Инвертировав аугментальную цепь *E-A=B-F*, мы добавим в паросочетание два ребра и удалим одно. Так образуется цепь *E=A-B=F* (здесь знаком «=» отмечено чёрное ребро паросочетания). В итоге в графе не осталось белых вершин, — процедура завершена.

Теперь рассмотрим порядок обработки графа, показанного на рис. 27-9. Здесь количество вершин нечётное, стало быть, одна из них после построения паросочетания неминуемо останется белой.

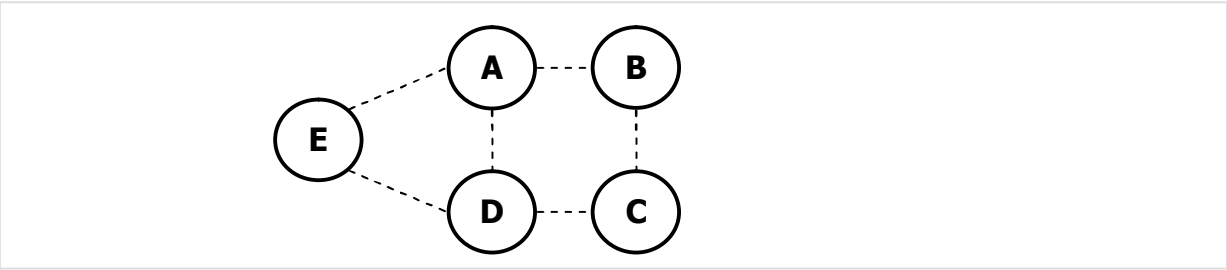


Рис. 27-9 — Граф с нечётным количеством вершин

Рёбра *A=B* и *C=D* будут добавлены в паросочетание точно так же, как описано выше. Затем, при построении дерева из вершины *E*, получится картина, показанная на рис. 27-10, — здесь из вершины *E* протянулись две нейтральные *венгерские* цепи, такое дерево тоже называют *венгерским*. В этой ситуации расширить паросочетание невозможно.

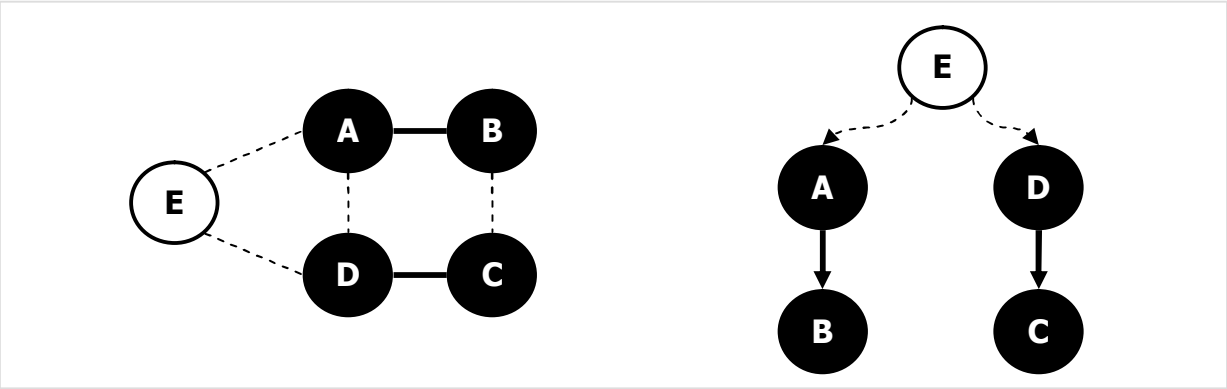


Рис. 27-10 — Дерево с корнем в C оказалось венгерским

Дадим теперь алгоритм построения *наибольшего* паросочетания в графе:

- Красим все вершины и рёбра белым.
- Для всех белых вершин:
  - Берём очередную белую вершину и строим (обходом в ширину) дерево с корнем в этой вершине. Постройка дерева завершается либо нахождением увеличивающей цепи, либо построением венгерского дерева.
  - Если на предыдущем шаге найдена увеличивающая цепь, инвертируем цвета рёбер вдоль этой цепи, и две крайние вершины цепи красим чёрным.

При постройке дерева учитываем, что ближняя к корню чёрная вершина пары является в этой паре *внутренней*, из неё по *чёрному* ребру можно следовать только в следующую чёрную вершину — *внешнюю*. А из *внешней* чёрной вершины по белым рёбрам пути открыты в другие вершины. Так достигается чередование белых и чёрных рёбер в создаваемых цепях.

Этот алгоритм применим и к двудольным графам. Отметим, что увеличивающая цепь всегда содержит нечётное количество рёбер или дуг. Поэтому, если для построения дерева выбирать белые вершины только из *левой* доли, то успешная постройка закончится в белой вершине *правой* доли. Соответственно конец *венгерской* цепи будет в чёрной вершине *левой* доли.

27.5. Венгерский алгоритм Куна

Итак, описанный выше алгоритм даёт *наибольшее* паросочетание, однако никак не учитывает веса рёбер. Для построения *оптимального* (минимального или максимального) паросочетания на *двудольном* графе используют *венгерский* алгоритм *Куна*, названый так в память о венгерских математиках, внёсших вклад в его разработку. Алгоритм Куна включает в себя описанный выше поиск *наибольшего* паросочетания, но вдобавок выполняет ещё ряд действий, смысл которых поясним следующим примером. Увы, наглядность примера сочетается с узостью набора данных, которым он может соответствовать.

Итак, вот страна, где по берегам Дуная приютилось *чётное* число городов, — по **N** городков на каждом берегу (для простоты условимся так). Страна известна странным и расточительным способом соединения *всех* города мостами так, как показано на рис. 27-11.

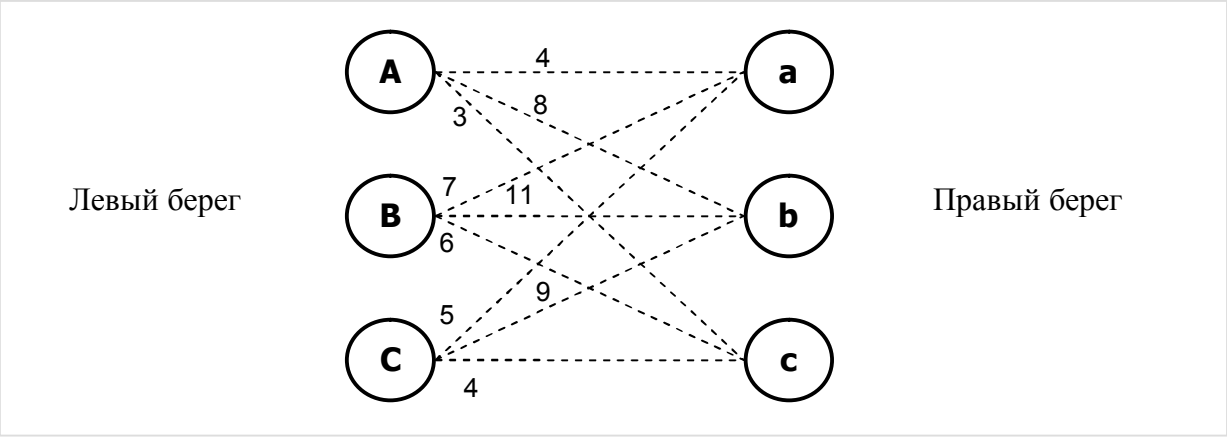


Рис. 27-11 — Схема городов с мостами (полный двудольный граф)

Здесь числа рядом с рёбрами относятся не к длине мостов, а характеризуют иное свойство — степень *наклона* моста, его *крутизну*. Дело в том, что правый берег реки выше левого, и все опоры правого берега выше левобережных, как это показано на рис. 27-12. Числа рядом с опорами указывают их высоту, а разность высот определяет *крутизну* моста, условно выражаемую *длиной* ребра графа.

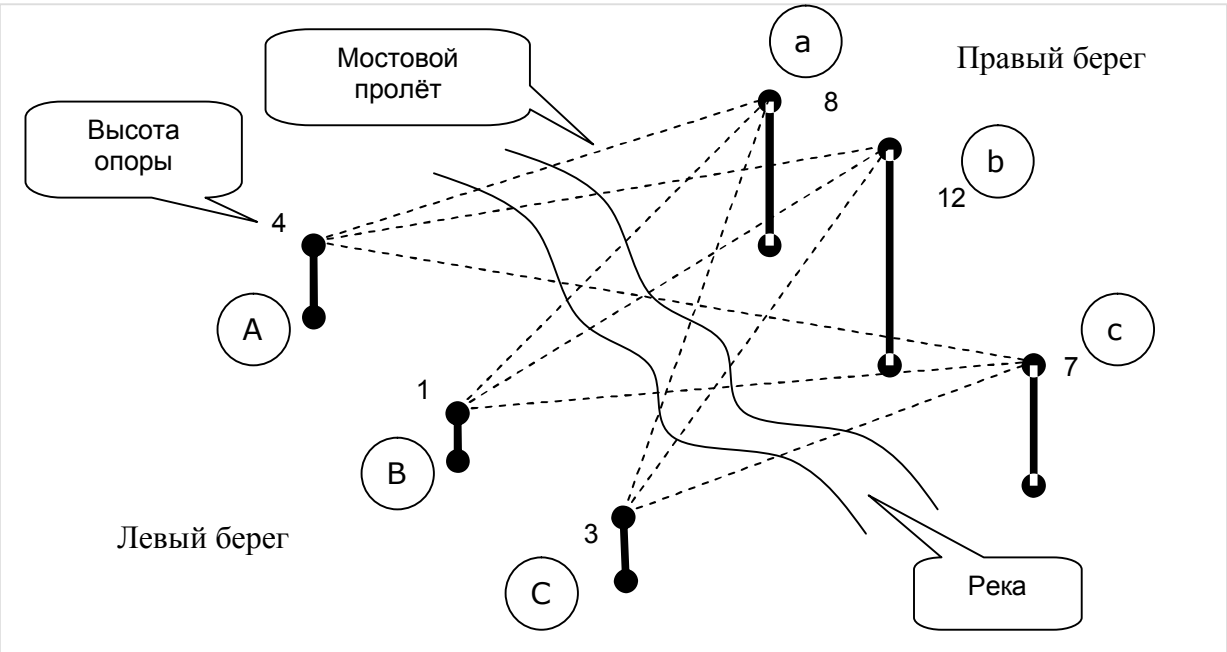


Рис. 27-12 — Левые и правые опоры мостов (правые опоры выше)

Ход технического прогресса навёл правительство на мысль соединить города железными дорогами. В целях экономии решили ограничиться минимум дорог с тем условием, чтобы каждый город с левого берега соединялся путями только с одним правобережным городом, а каждый правобережный — только с одним

левобережным. Легко догадаться, что мосты с путями здесь составят *наибольшее* паросочетание.

Но к чему тут уклон мостов? Дело в том, что по требованию путейцев перед прокладкой рельсов соответствующие пролёты мостов должны быть выровнены в горизонтальное положение. За выравнивание мостов взялась строительная фирма, которая тоже радела об экономии своих средств. Там решили, что будут выравнивать мосты поднятием части опор только в той степени, которая на выровненных мостах даст построить *наибольшее* паросочетание, а остальные не трогать. Теперь следим за действиями строителей.

Сначала они обследовали все опоры левого берега (белые вершины), пытаясь построить из них увеличивающие цепи и найти искомое паросочетание. Попытка не удалась, зато в ходе обследования выяснили, что, приподняв все левые опоры на 3 метра, можно гарантированно выровнять, по крайней мере, один мост. Из рисунка понятно, что для этого достаточно приподнять лишь опору *A*, но строители догадались, что прочие опоры в будущем поднимать тоже придётся. Итак, приподняв на три метра опоры *A*, *B* и *C* , строители выровняли мост *A-c*, и после второй попытки поиска паросочетания добавили к нему чёрное ребро *A=c* . Это состояние отражено на рис. 27-13, здесь отметим, что крутизна всех прочих мостов после поднятия опор тоже уменьшилась.

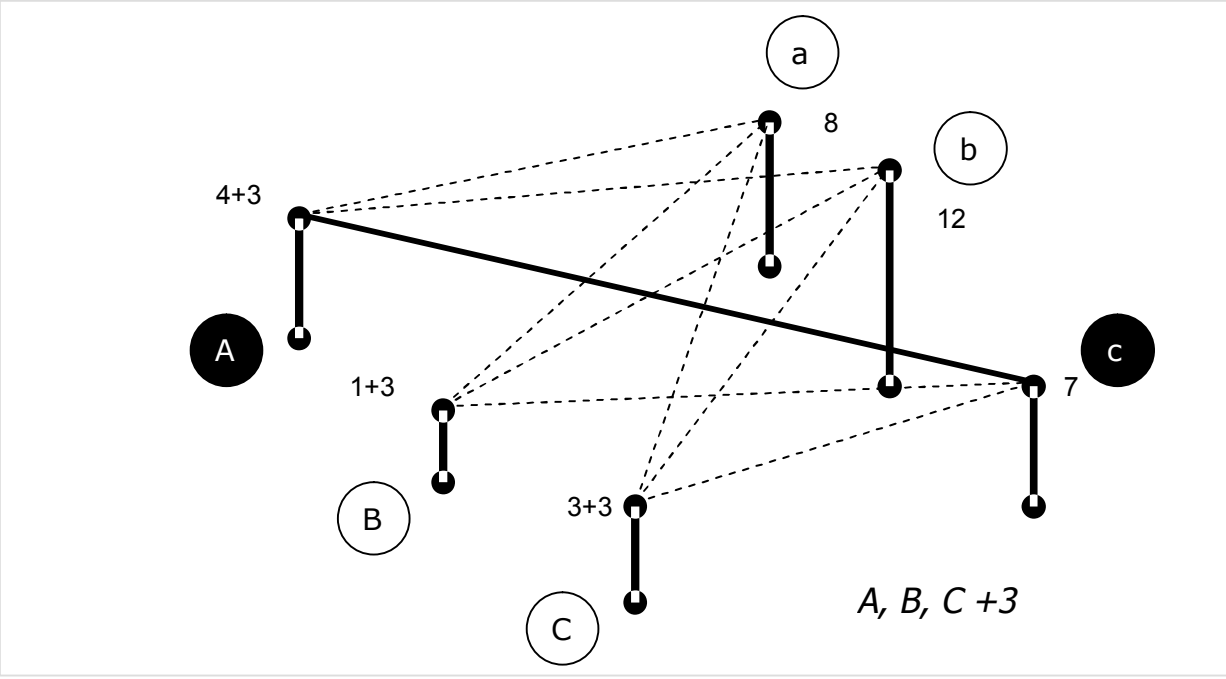


Рис. 27-13 — Поднятие опор на 3 метра и добавление ребра A=c в паросочетание

Затем строители предприняли третью попытку найти аугментальную цепь, начиная её из белых вершин *B* и *C*, — она вновь провалилась. Зато выяснилось, что для выравнивания следующего моста достаточно приподнять эти вершины на один метр, что и было сделано, результат показан на рис. 27-14, — здесь оказался выровненным мост *C-c* (горизонтальные мосты обозначаем сплошной линией).

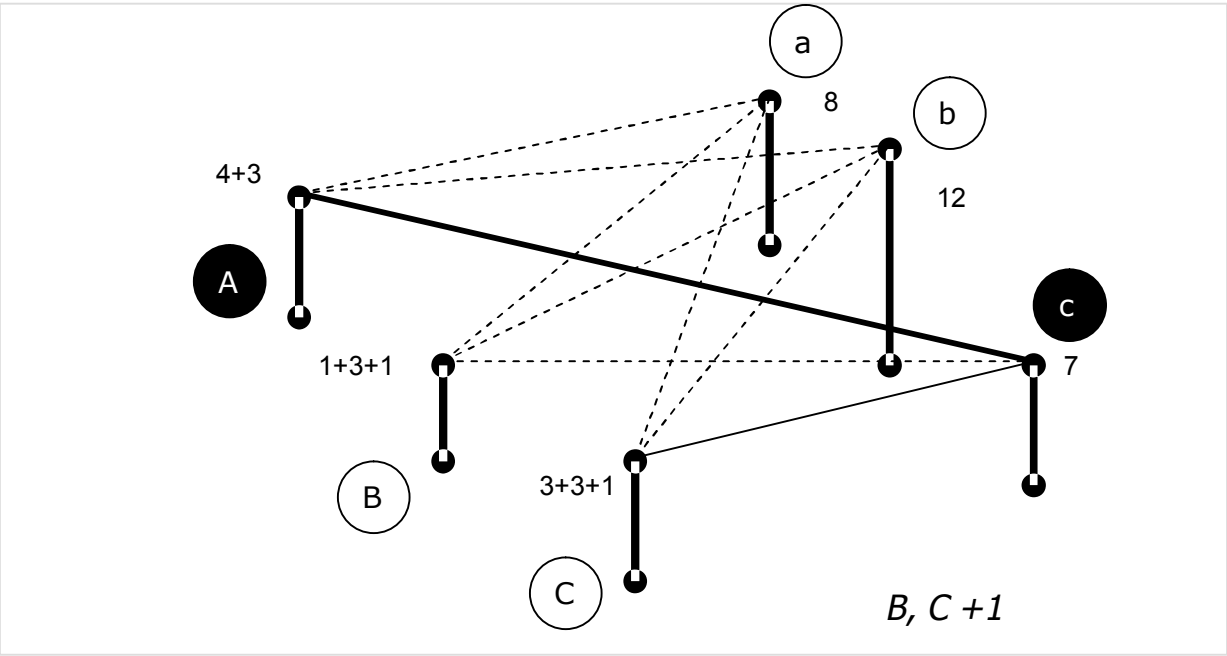


Рис. 27-14 — Поднятие опор B и C и на 1 метр и выравнивание моста C-c

Очередная попытка расширить паросочетание построением увеличивающих цепей из белых вершин *B* и *C* вновь оказалась неудачной: вершина *B* пока не связана с другими горизонтальными мостами, а из вершины *C* удалось построить только короткую венгерскую цепь *C-c=A* (знаком «=» метим чёрные ребра паросочетания). Теперь — внимание! — настало время ввести ещё одно понятие: *платформа* — так кратко назовём множество вершин, посещённых при попытке расширить паросочетание. До сих пор платформу составляли только *белые* вершины *левого* берега, но сейчас их география расширилась: в платформу вошли ещё две чёрные вершины с обоих берегов, в итоге множество посещённых вершин составило *A-B-C-c*.

Как всегда, неудачная попытка открыла строителям новое знание: для выравнивания очередного моста надо приподнять некоторые опоры ещё на метр. Но какие именно опоры? Разумеется, две белые на левом берегу, но... тогда нарушится горизонтальность моста *C-c*, следовательно, опора *c* тоже нуждается в подъёме. Однако, приподняв её, мы нарушим горизонтальность моста *A-c*. Так, следуя по цепочке горизонтальных мостов, строители поняли, что поднимать надо всю *платформу A-B-C-c*, что и было сделано (рис. 27-15). В результате выровнялись ещё два моста: *A-a* и *C-a*, что позволило в следующей *удачной* попытке добавить в паросочетание ребро *C=a*.

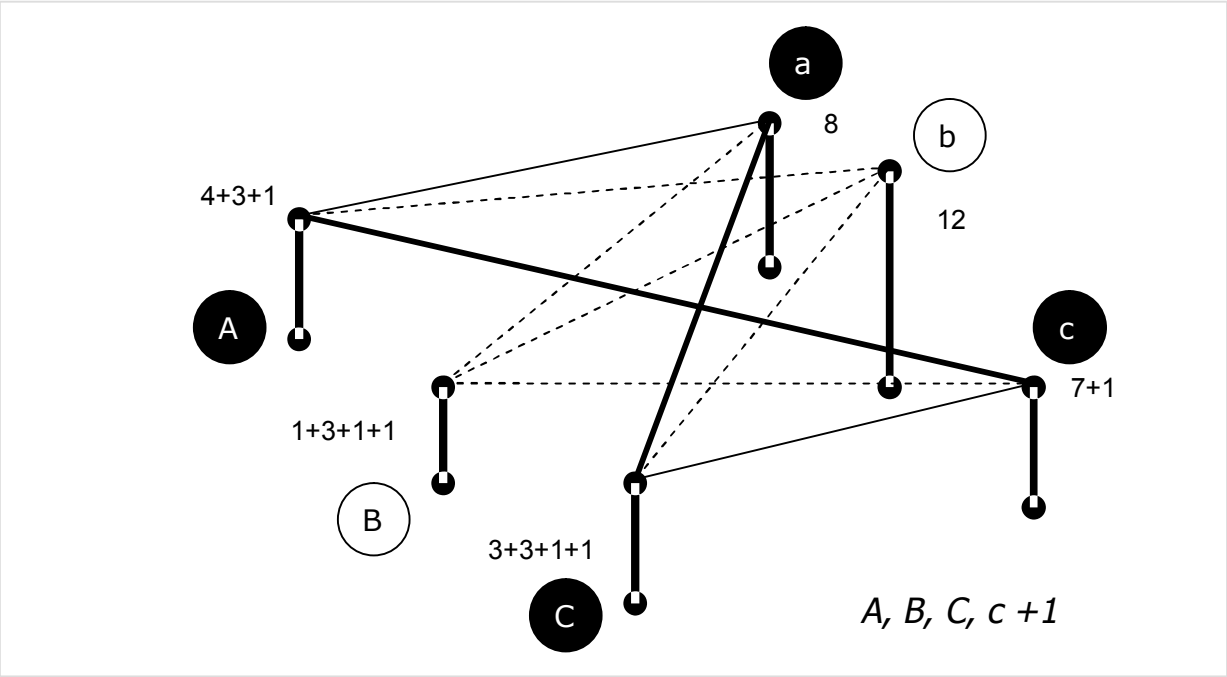


Рис. 27-15 — Поднятие на 1 метр платформы A-B-C-c и выравнивание мостов A-a и C-a, добавилось ребро C=a

Однако цель пока не достигнута — вершина *B* всё ещё белая, и потому строители попытались (неудачно) найти увеличивающую цепь из этой вершины. Тогда выяснили, что для выравнивания мостов *платформу* (а её составляет сейчас только вершина *B*) надо поднять на **2** метра. В итоге выровнялись ещё два моста: *B-a* и *B-c* (рис. 27-16).

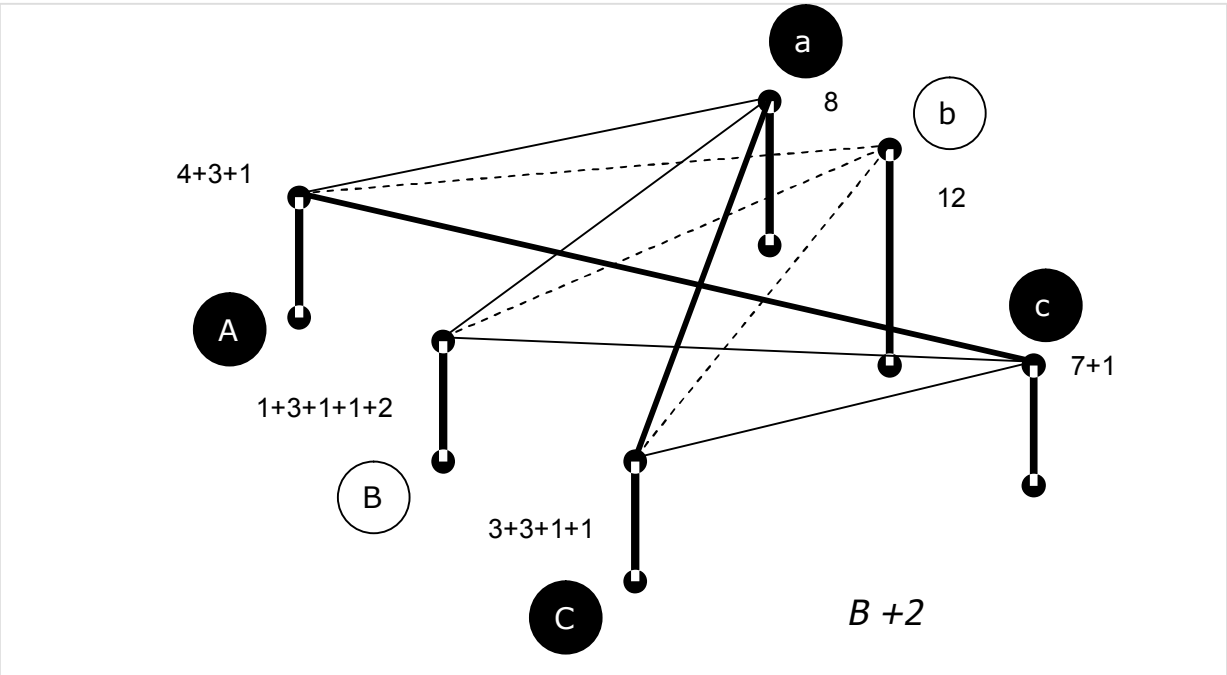


Рис. 27-16 — Поднятие опоры B и выравнивание мостов B-a и B-c

Следующая попытка расширить паросочетание вновь оказалась неудачной, но теперь по иной причине: из вершины *B* протянулись две нейтральные *венгерские*

цепи: *B-a=C-c=A* и *B-c=A-a=C*. Разобравшись, строители приподняли всю *платформу* (а здесь были посещены все вершины, кроме *b*) ещё на **4** метра, что добавило горизонтальный мост *B-b*. Легко догадаться, что следующая попытка расширить паросочетание оказалась успешной и завершающей (рис. 27-17).

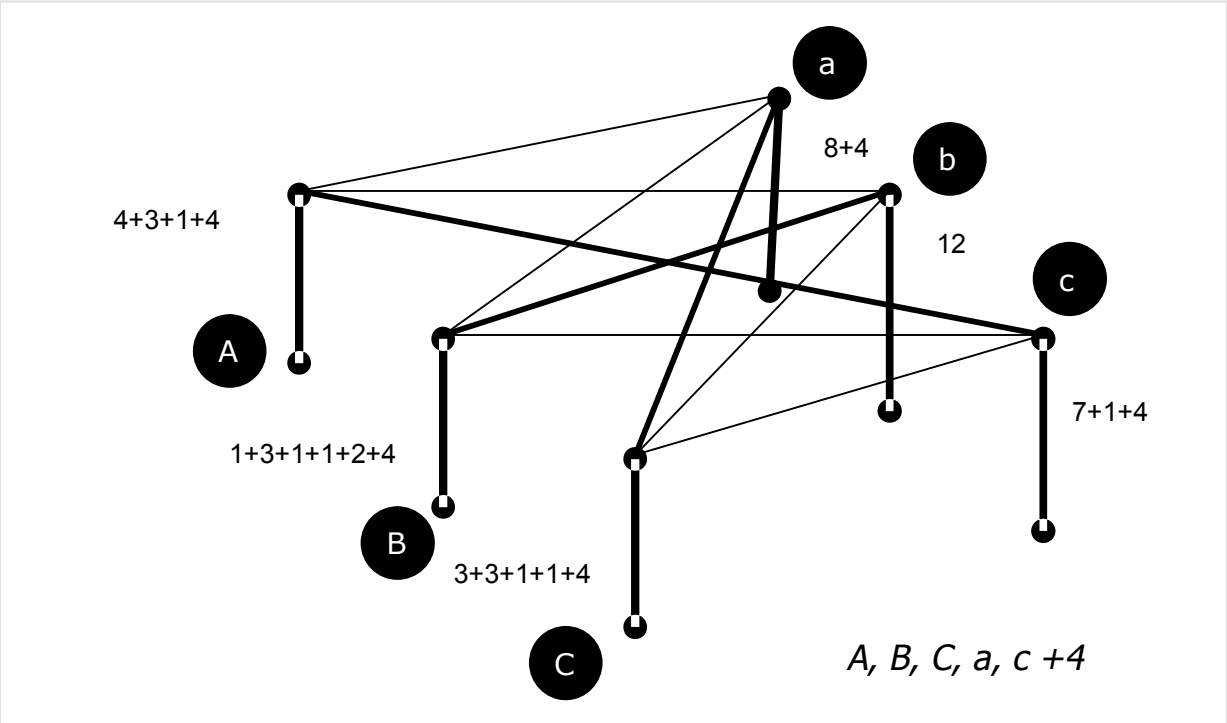


Рис. 27-17 — Поднятие платформы A-B-C-a-c на 4 метра и выравнивание моста B-b

Подведём промежуточный итог. После мудрёных манипуляций опоры мостов были приподняты до уровня самой высокой из них, и тогда все мосты выровнялись, и путейцы исполнили свою работу. Проницательный читатель давно понял: так можно было сделать сразу. Оправдаюсь тем, что сказка — ложь, да в ней намёк. Исходные данные здесь подобраны так, чтобы сработал наглядный мостовой пример, а он, как я предупреждал, не охватывает всё разнообразие графов, большинство из которых нельзя интерпретировать мостами. Однако математик докажет, что алгоритм *Куна* сработает на любых взвешенных двудольных графах, потому удостоим его здесь формального описания.

Но прежде от мостов снова перейдём к графу, и я напомню, что крутизна мостов тут соответствует длине рёбер. Тогда вершины уподобим опорам, и припишем им числовое поле **D** для обозначения их возвышения относительно исходного положения (в программе для этого служит поле **TNode.mDist**). Тогда текущая «крутизна пролёта» между вершинами **i** и **j** определится формулой:

Vij = V0ij – (Di–Dj), где

- **V0ij** — исходная крутизна пролёта (поле **TLink.mValue**);
- **Di** — текущее возвышение левой вершины **i;**
- **Dj** — текущее возвышение правой вершины **j**.

Напомню, что только рёбра с текущей *нулевой* крутизной участвуют в построении увеличивающих цепей. С учётом всего сказанного, алгоритм Куна можно представить так.

* Установить исходное состояние: все вершины и рёбра белые, поля возвышения **D** равны нулю.
* Пока не построено *наибольшее* паросочетание, повторять:
  - Снять признаки посещения всех вершин (очистить платформу).
  - Добавить в паросочетание все возможные рёбра, последовательно отыскивая увеличивающие цепи из всех белых вершин левой доли. В ходе поиска цепей отметить все *посещённые* вершины (платформу) и запомнить минимальное возвышение **Delta**, нужное для выравнивания очередного моста (возвышения платформы).
  - Если на предыдущем шаге рёбра не добавились, поднять платформу на высоту **Delta**, то есть, прибавить это число к полям возвышения **D** всех *посещённых* вершин.

Здесь вспомогательный алгоритм поиска увеличивающих цепей мы нагрузим двумя дополнительными обязанностями, что практически не усложнит его. Первая состоит в пометке всех посещённых вершин, а вторая — в запоминании минимальной крутизны тех рёбер, которые ещё не участвуют в построении цепей (имеют не нулевую текущую крутизну). В завершение пояснений дадим список основных терминов.

Табл. 27-1 — Основные термины, используемые для описания алгоритмов поиска паросочетаний

Термин	Пояснение
Паросочетание	**Подмножество рёбер такое, что никакие два из них не имеют общей (смежной) вершины.**
Максимальное **паросочетание**	***Паросочетание*****, которое нельзя расширить без удаления из него других рёбер. В графе может быть несколько максимальных паросочетаний.**
Наибольшее **паросочетание**	***Паросочетание*** **максимальной мощности, т.е., включающее в себя наибольшее количество рёбер. Наибольшее паросочетание является и максимальным.**
Совершенное **паросочетание**	***Паросочетание*****, в которое входят все вершины графа**
Оптимальное **паросочетание**	***Наибольшее*** **паросочетание на взвешенных рёбрах минимального либо максимального веса.**
Экспонированная **(свободная) вершина**	**Вершина, не принадлежащая паросочетанию.**
Альтернирующая цепь **(чередующаяся)**	**Простая цепь, ребра которой попеременно лежат и не лежат в некотором паросочетании.**
Аугментальная цепь **(увеличивающаяся)**	***Альтернирующая*** **(чередующаяся) цепь, начальная и конечная вершины которой экспонированы.**
Венгерская цепь **(нейтральная)**	***Альтернирующая*** **(чередующаяся) цепь, только начальная вершина которой экспонирована.**
Альтернирующее дерево **(чередующееся)**	**Дерево, построенное из корня, который является *экспонированной* (свободной) вершиной.**

27.6. Реализация алгоритма Куна

Перед рассмотрением метода, реализующего алгоритм Куна, опишем назначение полей вершин и рёбер в контексте венгерского метода:

- **TNode.mDist** — показывает степень «возвышения» вершины (опоры моста);
- **TNode.mColor** — текущий цвет вершины, чёрные вершины входят в паросочетание;
- **TNode.mFlow** — признак посещения вершины, то есть, признак принадлежности её к платформе (0 или 1);
- **TNode.mRoot** — обратная ссылка (линк) в построенном дереве (ссылка в направлении корня);
- **TNode.mLink** — чёрный линк паросочетания для этой вершины (**nil**, если вершина белая);
- **TLink.mValue** — исходная крутизна ребра (мостового пролёта);
- **TLink.mColor** — текущий цвет ребра, чёрные входят в паросочетание;
- **TLink.mLow** — остаточная крутизна ребра по завершении метода;
- **TLink.mHigh** — признак участия ребра в данном алгоритме, 0 — ребро активно, а иначе считается отключенным.

Последние два элемента перечня поясним особо. Дело в том, что алгоритм Куна нередко входит в состав других, например, в алгоритм поиска кратчайшего гамильтонова контура, — это тема 32-й главы. И там от Куна попросят дополнительных данных, а именно остаточную крутизну ещё не выправленных мостов. Эту крутизну мы сохраним по выходе из метода в полях **TLink.mLow**. Кроме того, некоторые рёбра (или дуги) будут временно изыматься из графа, отключаться. С этой целью поле **TLink.mHigh** будет использовано как признак отключения ребра: 0 — ребро активно, используется; 1— ребро пассивно, отключено. По умолчанию это поле очищено, и все рёбра сейчас активны.

Основной метод, реализующий венгерский алгоритм Куна, назван **MarkMinPairsDicoty**, он метит чёрным рёбра и вершины *наибольшего* паросочетания *минимальной* стоимости, вычисляет эту стоимость, а также сохраняет остаточную крутизну рёбер в полях **TLink.mLow**.

Листинг 27-1 — Основной метод, реализующий венгерский алгоритм Куна

```
function TGraph.MarkMinPairsDicoty(var aPairs: integer): integer;
var    Que: TBuffer;  // очередь вершин
    NodesL: TSet;     // множество левых вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Выполняется начальная установка двудольного графа:
  // все вершины и дуги белые
  // Формируется вспомогательное подмножество левых вершин
  // Возвращает максимально возможное количество пар

  function  InitNodes: integer;
  var Node: TNode;
```

```
    Link: TLink;
    L, R: integer; // счётчики левых и правых вершин
begin
  // Обнуление счётчиков вершин
  L:= 0; R:= 0;
  // Перебор всех вершин:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mColor:= CWhite;   // цвет вершины белый
    if Node.IsLeft then begin
      Inc(L);               // подсчёт левых вершин
      Node.mDist:= MaxInt;  // вершинное число ищем вторым перебором
      NodesL.Insert(Node);  // и вставка в множество левых
    end else if Node.IsRight then begin
      Inc(R);               // подсчёт правых вершин
      Node.mDist:=0;        // вершинное число = 0
    end;
    Node:= NodeNext;
  end; // while
  // Количество ожидаемых паросочетаний равно меньшему из двух:
  if L > R then Result:= R else Result:= L;
  // Перебор левых вершин:
  Node:= NodesL.GetFirst as TNode;
  while Assigned(Node) do begin
    // Перебор исходящих линков
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do if mHigh=0 then begin
        // это активный линк
        Link.mColor:= CWhite;
        // запоминаем вес минимальной дуги
        if Node.mDist > Link.mValue  then
          Node.mDist:= Link.mValue; // начальное "приподнятие опоры"
      end;
      Link:= Node.OutLinkNext;
    end; // while
    Node:= NodesL.GetNext as TNode;
  end; // while
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск аугментальной (улучшающей) цепи:
// aLeft - корневая экспонированная левая вершина дерева (белая)
// Возвращает экспонированную правую вершину (белую),
// при условии, что найдена аугментальная цепь, а иначе NIL.
// Формирует ссылки в полях TNode.mRoot, ведущие к корню дерева.
// Если экспонированная вершина не найдена,
// то параметр aDelta содержит минимальное приращение цены,
// необходимое для покупки ещё хотя бы одного линка.

function FindAugmenting(aLeft : TNode; var aDelta: integer): TNode;
var Node: TNode;
    Link: TLink;
    Cost: integer;
begin
  Result:= nil;        // ссылка на правую вершину
  Que.Clear;           // очистить очередь вершин
  Que.Put(aLeft);      // и поместить туда корневую вершину
  // Пока очередь не пуста и не обнаружена аугментальная цепь
  while (Que.GetCount > 0) and not Assigned(Result) do begin
    Node:= Que.Get as TNode;
    Node.mFlow:= 1;    // отмечаем посещение вершины
    if Node.IsLeft then begin
```

```
        // Это вершина левой доли
        // Ищем белую вершину в правой доле, а чёрные ставим в очередь
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // обрабатываем только активные белые линки
          with Link do if (mHigh=0) and    // активный линк
                          (mColor=CWhite)  // и линк белый
            then begin
              // Вычисляем приведенную (пониженную) стоимость дуги
              Cost:= mValue - (mOwner.mDist - mDest.mDist);
              if Cost = 0 then begin
                // Это достижимая правая вершина (Cost=0)
                if  mDest.mColor = CWhite then begin
                  // Если она белая,
                  // то найдена белая вершина и аугментальная цепь
                  Result:= mDest;      // экспонированная белая вершина
                  Result.mRoot:= Link; // ссылка в направлении корня дерева
                  Break;               // выход из цикла и процедуры
                end else if mDest.mFlow = 0 then begin
                  // Найдена чёрная вершина,
                  // она ещё не посещалась, и потому ставим её в очередь
                  mDest.mRoot:= Link; // ссылка в направлении корня дерева
                  Que.Put(mDest);     // ставим в очередь
                end; // else
              end else begin
                // Это недостижимая правая вершина (Cost > 0),
                // через aDelta возвращаем минимальное приращение цены
                if aDelta > Cost then aDelta:= Cost;
              end;
          end; // with Link
          Link:= Node.OutLinkNext;
        end; // while - конец перебора линков
      end else begin
        // Это внутренняя чёрная вершина в правой доле (Node.mIsLeft = FALSE)
        Link:= Node.mLink;         // извлекаем из неё линк паросочетания
        Link.mOwner.mRoot:= Link; // это ссылка в направлении корня дерева
        // Ставим в очередь найденную внешнюю чёрную вершину в левой доле
        Que.Put(Link.mOwner);
      end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инверсия аугментальной (увеличивающей) цепи.
  // Вызывается при условии, что такая цепь обнаружена
  // в процедуре FindAugmenting.
  // aLeft, aRight -- левая и правая экспонированные (белые) вершины.
  // Использует поля Node.mRoot, сформированные процедурой FindAugmenting,
  // и формирует из них линки паросочетаний Node.mLink.

  procedure Inverse(aLeft, aRight : TNode);
  var Node: TNode;
      Link: TLink;
      Flag: boolean;
  begin
    // Проходим аугментальную цепь в обратном порядке
    // инвертируя цвета дуг
    Flag:= true;      // инвертируемый флаг определяет цвет ребра (дуги)
    Node:= aRight;
    repeat
      Link:= Node.mRoot as TLink;
      if Flag then begin
        // Переход справа налево, окраска дуги чёрным
```

```
      // Красим белым старый линк паросочетания
      with Node do if Assigned(mLink) then mLink.mColor:= CWhite;
      Node.mLink:= Link;    // запоминаем новый линк паросочетания
      Link.mColor:= CBlack; // и красим его чёрным
      Node:= Link.mOwner;   // переход в сторону корня дерева
      Node.mLink:= Link;    // и здесь запоминаем новый линк паросочетания
    end else begin
      // Переход слева направо, окраска дуги белым
      Link.mColor:= CWhite;
      Node:= Link.mDest;     // переход в сторону корня дерева
    end;
    Flag:= not Flag; // инвертируемый флаг определяет цвет ребра (дуги)
  until Node = aLeft;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Очистка у всех вершин полей TNode.mFlow -- признаков посещения.
// Вызывается перед обходом свободных (белых) левых вершин

procedure ClearFlags;
var Node: TNode;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mFlow:= 0;   // Это признак посещения при постройке деревьев
    Node:= NodeNext;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Обход свободных (белых, экспонированных) левых вершин
// с целью построения улучшающих аугментальных цепей.
// Возвращает количество обнаруженных цепей (и соответственно пар).
// Если ни одна такая цепь не будет обнаружена,
// то через параметр aDelta возвращает минимальное "поднятие опоры",
// необходимое для выпрямления хотя бы одного моста

function FindPairs(var aDelta: integer): integer;
var LNode, RNode : TNode;
begin
  Result:= 0;        // счётчик обнаруженных пар
  aDelta:= MaxInt;   // минимальное приращение цены
  ClearFlags;        // очистка признаков посещения вершин TNode.mFlow
  // Перебор левых вершин
  LNode:= NodesL.GetFirst as TNode;
  while Assigned(LNode) do begin
    if LNode.mColor = CWhite then begin
      // Это открытая (экспонированная) левая вершина
      // Ищем открытую вершину справа и формируем aDelta
      RNode:= FindAugmenting(LNode, aDelta);
      if Assigned(RNode) then begin
        // Аугментальная цепь найдена
        // Инвертируем цвета дуг от RNode к LNode
        Inverse(LNode, RNode);
        // Две крайние вершины цепи метим чёрным
        // как вошедшие в паросочетание
        LNode.mColor:= CBlack;
        RNode.mColor:= CBlack;
        Inc(Result);  // и наращиваем счётчик пар
      end;
    end;
    LNode:= NodesL.GetNext as TNode;
  end;
end;
```

```
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура корректирует вершинные числа Node.mDist на величину aDelta
  // (приподнимает платформу)
  // Вызывается после обхода экспонированных (белых) вершин
  // после того, как ни одна новая пара не обнаружена.

  procedure Correct(aDelta: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Если вершина посещалась, то нарастить вершинное число
      if Node.mFlow <> 0 then begin
        Inc(Node.mDist, aDelta);
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт стоимости паросочетания по сумме чёрных линков
  // и формирование пониженной стоимости (крутизны) дуг TLink.mLow
  // с учётом текущих значений вершинных чисел TNode.mDist.
  // (поля TLink.mLow используются при поиске цикла Гамильтона)

  function CalcCost: integer;
  var Node: TNode;
      Link: TLink;
  begin
    Result:= 0;
    // Перебор вершин левой доли
    Node:= NodesL.GetFirst as TNode;
    while Assigned(Node) do begin
      // Node.mLink - это чёрный линк паросочетания
      Inc(Result, Node.mLink.mValue);
      // Перебор исходящих линков
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // если это активный линк, вычисляем редуцированную стоимость
        with Link do
          if mHigh=0 then mLow:= mValue - (mOwner.mDist - mDest.mDist);
        Link:= Node.OutLinkNext;
      end;
      Node:= NodesL.GetNext as TNode;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
   P: integer;     // количество новых пар на очередной итерации
   Delta: integer; // приращение, необходимое для приподнятия платформы
   Count: integer; // счётчик осташихся не найденных пар

begin  { TGraph.SetPairsDicoty }

  Que:= TBuffer.Create; // очередь вершин
  NodesL:= CreateSet;   // множество левых вершин

  Count:= InitNodes; // возвращает максимально возможное число пар
  aPairs:= Count;

  repeat     // повторять до формирования всех паросочетаний
    repeat   // повторять, пока возможно приращение паросочетания
      P:= FindPairs(Delta); // P - количество обнаруженных новых пар
```

```
                              // Delta - миним. приращение для покупки дуги
      Dec(Count, P);          // Count - счётчик оставшихся пар
    until (P=0) or (Count=0);
    // Если паросочетание не полно, то скорректировать вершинные числа
    // на величину Delta (приподнять платформу),
    // "выровняв" тем очередной мост или несколько мостов
    if Count > 0 then Correct(Delta);
  until Count=0;

  // Здесь максимальное паросочетание достигнуто,
  // подсчитать стоимость паросочетания и остатки стоимостей дуг
  Result:= CalcCost;
  // Очистка памяти:
  NodesL.Free;  // множество левых вершин
  Que.Free;     // очередь вершин
end;
```

Отметим здесь несколько моментов. В функции инициализации **`InitNodes`** формируется вспомогательное множество левых вершин, что в последующем ускоряет их перебор. Принадлежность вершины к левой доле графа определяется методом **`TNode.IsLeft`**:

```
function TNode.IsLeft: boolean;
begin
  Result:= mLnkOut.GetCount <> 0;
end;
```

То есть, к вершинам левой доли причислены те, из которых исходит хотя бы одна дуга. Разумеется, что это работает только в ориентированном двудольном графе, и в иных случаях может потребоваться другой признак отнесения к долям. Затем определяем количество пар, которое составит *наибольшее* паросочетание. Это будет меньшее из двух: число вершин либо в левой, либо в правой долях.

Там же — при инициализации — производим начальный «подъём мостовых опор»: вершинным числам **`mDist`** в левой доле присваиваем минимальные значения так, чтобы гарантировано «выровнять» хотя бы по одному «мосту», исходящему из левых вершин. Действительно, это условие необходимо для получения наибольшего паросочетания, хотя и не достаточно. Начальное «выравнивание» уменьшает количество последующих итераций.

Следующая функция (метод) формирует объект, содержащий множество дуг *наибольшего* паросочетания *минимальной* стоимости и его стоимость. Здесь предварительно вызывается упомянутый выше метод **`MarkMinPairsDicoty`**.

Листинг 27-2 — Метод, формирующий множество дуг паросочетания и его стоимость

```
function TGraph.GenMinPairsDicoty: TCostSet;
var Cost: integer;    // стоимость паросочетания
    Pairs: integer;   // количество найденных пар
    Node: TNode;
    S : TSet;
begin
  Cost:= MarkMinPairsDicoty(Pairs);
  S:= CreateSet;
```

```
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft
      then S.Insert(Node.mLink);
    Node:= NodeNext;
  end;
  Result:= TCostSet.Create(Cost, S, true);
end;
```

Испытаем метод после знакомства с ещё одним алгоритмом, решающим ту же задачу через поток.

27.7. Альтернативное решение через поток

Простота второго метода поиска паросочетания обусловлена использованием уже известного нам алгоритма нахождения насыщенного потока минимальной стоимости (см. главу 26). Идея метода в следующем.

Дополним двудольный граф двумя вершинами: искусственным истоком *S*, и искусственным стоком *T*, и соединим их соответственно с левой и правой долями дугами *нулевой* стоимости так, как показано на рис. 27-18. Всем дугам получившегося графа припишем неделимую *единичную* пропускную способность.

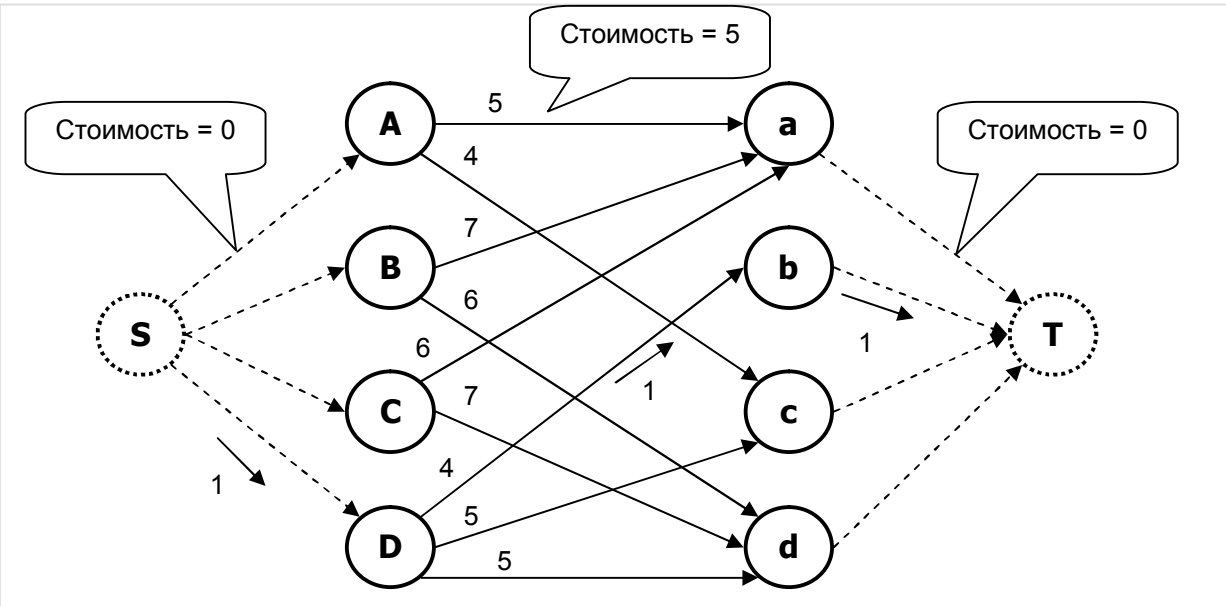


Рис. 27-18 — Дополнение графа вспомогательным истоком S и стоком T, пропускная способность всех дуг равна единице

Найдём здесь распределение наибольшего потока минимальной стоимости. Понятно, что каждая вершина *левой* доли сможет принять от источника *S* и передать дальше только единицу потока (поток не дробится). То же относится и к вершинам *правой* доли: исходящие из них дуги способны отправить в сток *T* лишь по одной единице потока. В дугах исходного графа поток распределится по «дешёвым» дугам, — они то и составят оптимальное в данном случае паросочетание. Отметим, также, что наибольший поток насытит вспомогательные

дуги либо истока, либо стока, либо того и другого сразу (если число вершин в долях одинаково).

В следующем листинге дан метод поиска наибольшего паросочетания минимальной стоимости на двудольном графе с применением потокового алгоритма.

Листинг 27-3 — Поиска наибольшего паросочетания минимальной стоимости через поток

```
function TGraph.GenMinPairsDicotyFlow: TCostSet;
var  S, T : TNode;
     Node : TNode;
     Link : TLink;
     Cost : integer;
      Cnt : integer;
      Res : TSet;
begin
  // Всем дугам графа назначаем единичную пропускную способность
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        mTemp:= mHigh;  // временно сохраняем
        mHigh:= 1;      // и устанавливаем единичную пропускную способность
      end;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
  // Создаём искусственные исток S и сток R:
  S:= TNode.Create(0, Self);
  T:= TNode.Create(0, Self);

  // Перебираем вершины, соединяя левую долю с истоком, а правую со стоком
  // дугами со стоимостью =0, пропускной способностью =1
  // Одновременно подсчитываем вершины в левой доле
  Cnt:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft then begin
      // Дуга из истока в левую долю,
      Link:= S.MakeLink(Node, 0);
      Link.mHigh:= 1;
      Inc(Cnt);
    end  else begin
      // Дуга из правой доли в сток,
      Link:= Node.MakeLink(T, 0);
      Link.mHigh:= 1;
    end;
    Node:= NodeNext;
  end;  // while

  // Из количества вершин в левой и правой долях выбираем меньшее:
  if Cnt > mNodes.GetCount div 2 then Cnt:= mNodes.GetCount - Cnt;

  // Вставляем в граф:
  InsertNode(S);   // исток
  InsertNode(T);   // сток
```

```
  // Находим поток минимальной стоимости:
  Cost:= CalcMinCostFlow(S, T, Cnt);

  // Удаляем и уничтожаем искусственные исток и сток:
  RemoveNode(T);
  RemoveNode(S);
  T.Free; S.Free;

  Res:= CreateSet;
  if Cost>=0 then begin
    // Здесь паросочетание существует
    // В паросочетание включаем дуги с ненулевым потоком (Link.mFlow <> 0)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          if mFlow <> 0 then Res.Insert(Link);
          mHigh:= mTemp; // восстанавливаем исходные значения
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end; // if
  // Формируем множество со стоимостью:
  Result:= TCostSet.Create(Cost, Res, true);
end;
```

27.8. Испытание методов

Прежде всего, сравним результаты, выдаваемые двумя рассмотренными выше методами поиска паросочетания минимальной стоимости на двудольном графе, этой цели служит следующая программа.

Листинг 27-4 — Программа для сравнения результатов двух методов поиска паросочетания минимальной стоимости

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TCostSet;   // результат
    Nodes : integer;  // количество вершин

// Вывод результата в текстовый файл

procedure ResultExpo(const aFile: String);
var Pair : TItem;
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
```

```
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('----------------------');
  with Res do begin
    if mCost<0 then Exit;
    Pair:= mSet.GetFirst;
    while Assigned(Pair) do begin
      if Pair is TLink then
        with Pair as TLink do Writeln(mOwner.GetName + '-'
                                      + mDest.GetName, ' =', mValue:3)
        else begin Pair.Expo; Writeln end;
      Pair:= mSet.GetNext;
    end;
    Writeln('Cost= ', mCost);
  end;
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;

const CFile = 'Out.txt';

begin
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Nodes < 3 then Break;
      // Создаётся случайный двудольный граф
      Gr:= THugeGraph.GenDicoty(99, Nodes);
      { Либо вводится из текстового файла
      Gr:= TGraphChars.Load('Dicoty-8.txt');
      }
      // Решение через поток:
      Res:= Gr.GenMinPairsDicotyFlow;
      ResultExpo('');  ResultExpo(CFile);
      Res.Free;

      // Решение алгоритмом Куна:
      Res:= Gr.GenMinPairsDicoty;
      ResultExpo('');  ResultExpo(CFile);
      Res.Free;
      Gr.Free;
    until false;
end.
```

На графе, текстовое представление которого дано ниже,

```
Dicoty-8.txt (Kristofides, 408)
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
16 - количество вершин
A B C D E F G H a b c d e f g h
A -> a=13 b=21 c=20 d=12 e= 8 f=26 g=22 h=11
B -> a=12 b=36 c=25 d=41 e=40 f=11 g= 4 h= 8
C -> a=35 b=32 c=13 d=36 e=26 f=21 g=13 h=37
D -> a=34 b=54 c= 7 d= 8 e=12 f=22 g=11 h=40
E -> a=21 b= 6 c=45 d=18 e=24 f=34 g=12 h=48
F -> a=42 b=19 c=39 d=15 e=14 f=16 g=28 h=46
G -> a=16 b=34 c=38 d= 3 e=34 f=40 g=22 h=24
H -> a=26 b=20 c= 5 d=17 e=45 f=31 g=37 h=43
```

обоими методами получено паросочетание стоимостью **76** единиц:

```
A-a = 13
B-h =  8
C-g = 13
D-e = 12
E-b =  6
F-f = 16
G-d =  3
H-c =  5
```

Отметим здесь, что в общем случае методы могут дать разные паросочетания *одинаковой* стоимости.

Теперь сравним быстродействие методов, воспользовавшись следующей программой.

Листинг 27-5 — Программа для сравнения быстродействия методов поиска паросочетания

```
{$APPTYPE CONSOLE}
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
  Gr : TGraph;
  Res : TCostSet;    // результат (паросочетание)
  Start: TDateTime; // для засечки времени
  Time : integer;
  Nodes : integer;  // количество вершин
  Retry : integer;  // количество повторений
  i : integer;
  Sum1, Sum2 : integer;

// Вывод результатов

procedure ResultExpo(const aFile: String);
var n: integer;
```

```
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('-------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4);
  Writeln('Flow= ', Sum1/Retry:7:1, ' ms');
  Writeln('Hung= ', Sum2/Retry:7:1, ' ms');
  Writeln('F/H = ', (Sum1+0.1)/(Sum2+0.1) :7:1);
  Writeln('-------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;

begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Nodes<3 then Break;
    Write('Retry= '); Readln(Retry);
    if Retry=0 then Break;
    Sum1:= 0; Sum2:= 0;
    for i:= 1 to Retry do begin
      Gr:= THugeGraph.GenDicoty(99, Nodes);
      Write('.');
      // Через поток:
      Start:= Now;
      Res:= Gr.GenMinPairsDicotyFlow;
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum1, Time);
      Res.Free;
      // Венгерским
      Start:= Now;
      Res:= Gr.GenMinPairsDicoty;
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      Gr.Free;
    end;
    Writeln(#7);  ResultExpo(''); ResultExpo('Out.txt');
  until false;
end.
```

В эксперименте порождены по сотне случайных графов размерностью 50, 100, 150 и 200 пар вершин. В табл. 27-2 даны средние времена их обработки.

Табл. 27-2 — Результаты сравнения быстродействия двух методов поиска паросочетания

Количество пар вершин	Среднее время обработки, ms		Отношение времён
	Через поток	Методом Куна	
50	5,7	0,8	6,9
100	32,4	2,9	11,4
150	99,1	5,9	16,8
200	229,6	10,6	21,7

Следующий рисунок показывает линейный рост скоростного преимущества венгерского метода Куна в сравнении с потоковым методом. Из этих результатов следует важный для практики вывод: для вычисления оптимального потока (минимальной либо максимальной стоимости) в специальном двудольном графе вместо стандартного метода можно воспользоваться более быстрым алгоритмом Куна. Этой мыслью мы воспользуемся в своё время при поиске кратчайшего гамильтонова контура.

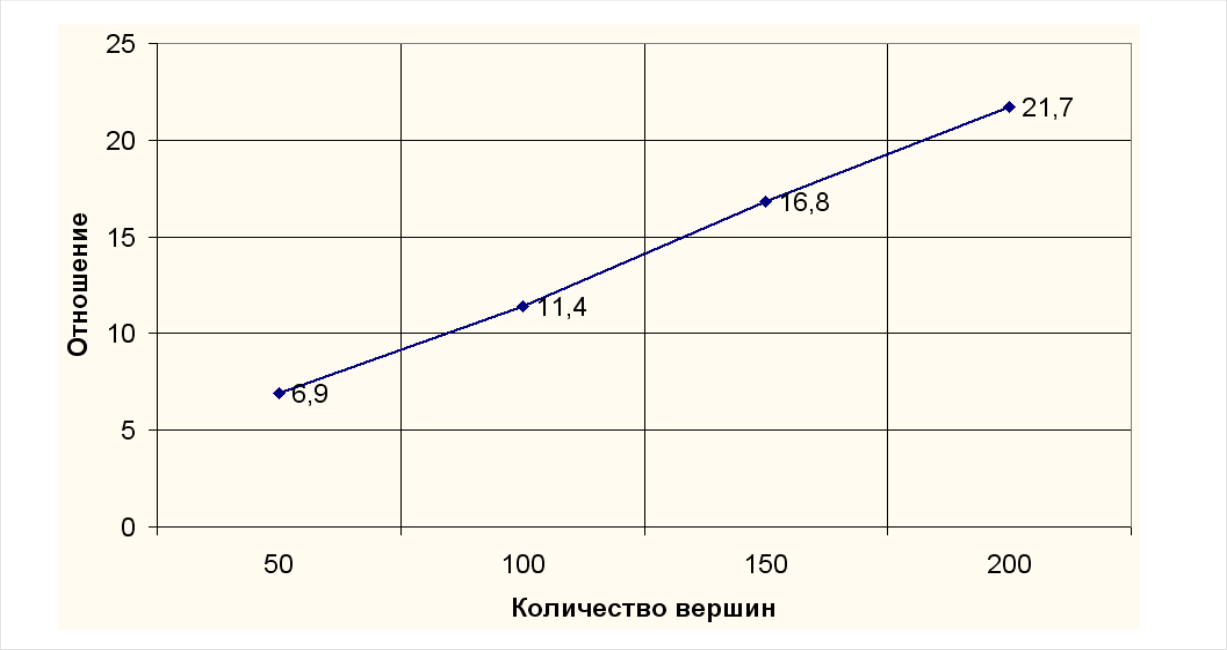


Рис. 27-19 — Соотношение быстродействия потокового метода и метода Куна в зависимости от размера двудольного графа

27.9. Итоги

27.9.1. Задачи поиска паросочетаний состоят в том, чтобы найти либо наибольшее паросочетание в графе — без учёта веса рёбер, либо какое-то из оптимальных: минимального или максимального веса.

27.9.2. Быстрые алгоритмы поиска паросочетаний основаны на поиске увеличивающих (аугментальных) цепей.

27.9.3. Венгерский алгоритм Куна ищет оптимальные паросочетания на двудольных графах. Он использует принцип поиска увеличивающих цепей в комбинации с накоплением особых вершинных чисел.

27.9.4. Эта же задача на двудольном графе может решаться потоковым алгоритмом, но существенно медленней.

27.9.5. Ввиду эквивалентности двух методов поиска паросочетания на двудольном графе, поиск оптимального потока в нём можно выполнить более быстрым алгоритмом Куна.

27.10. Задания

27.11. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 405
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 178
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 28
Паросочетание в произвольном графе

Предыдущая глава познакомила читателя с двудольными графами и быстрым венгерским алгоритмом Куна для поиска паросочетаний в них. Двудольными графами моделируются асимметричные связи между разнородными объектами, а это не исчерпывает всех отношений в реальном мире. Рассмотрим пример, где надо создавать пары из однородных равноправных элементов.

Пусть имеются **N** сотрудников, и технология производства требует их парной работы. Соединяя работников в пары, руководитель хочет вовлечь наибольшее число людей, и учесть, насколько они «сыгранны», могут ли работать вместе, и насколько эффективно. Ситуация моделируется произвольным графом, где ребро означает факт взаимной совместимости сотрудников, а вес ребра отражает эффективность этой пары. Тогда для лучшего суммарного эффекта руководителю следует искать в таком графе *наибольшее* паросочетание *максимального* веса.

В иных ситуациях целью может быть *наибольшее* паросочетание *минимального* веса, или *наибольшее* паросочетание без учёта веса рёбер. Легко догадаться, что все эти случаи можно свести к поиску паросочетания минимального веса. Для решения задачи разработан быстрый алгоритм Эдмондса, основанный на отыскании увеличивающих цепей. Здесь познакомимся с менее эффективным, но более простым методом. К его достоинствам относится и некая универсальность: так, после небольшой модификации можно находить не только паросочетания, но и сочетания из **M** связанных вершин (для объединения сотрудников в бригады из **M** человек).

28.1. Постановка задачи

Прежде всего, уточним, какие рёбра или дуги графа мы будем отыскивать. Рассмотрим произвольный граф на рис. 27-1.

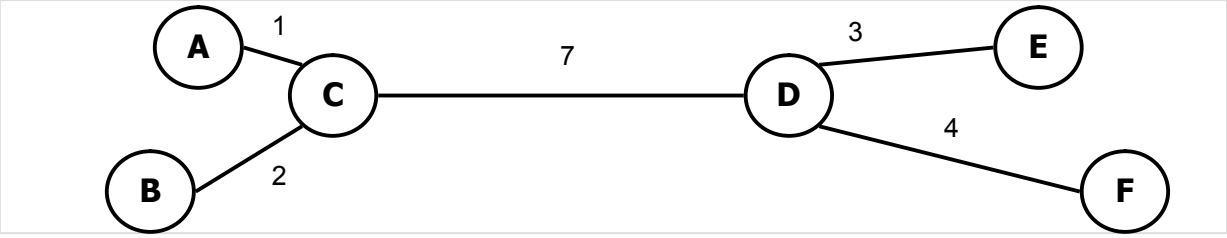


Рис. 28-1 — Граф с взвешенными рёбрами

Без учёта веса рёбер здесь можно найти четыре *наибольших* паросочетания:

```
1) A-C, D-E
2) A-C, D-F
3) B-C, D-E
4) B-C, D-F
```

Одно из них представляет собой паросочетание *минимального* веса (далее подразумеваем только *наибольшие* паросочетания), другое — паросочетание *максимального* веса (см. рис. 28-2 и рис. 28-3).

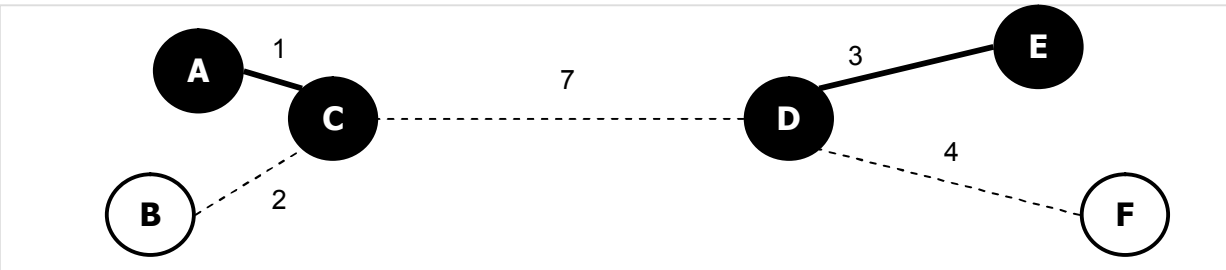


Рис. 28-2 — Наибольшее паросочетание минимального веса (1+3)

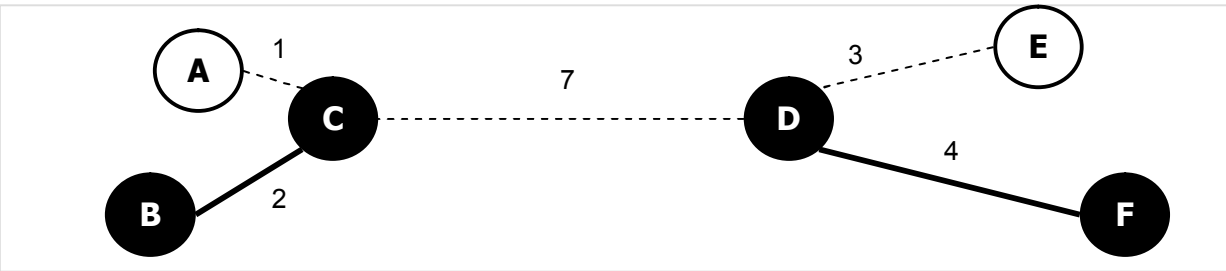


Рис. 28-3 — Наибольшее паросочетание максимального веса (2+4)

Отметим здесь, что отдельное ребро *C-D* тоже является паросочетанием, причём вес его максимален (7 против 6). Однако это паросочетание *не является наибольшим*, и потому не будет целью будущего алгоритма.

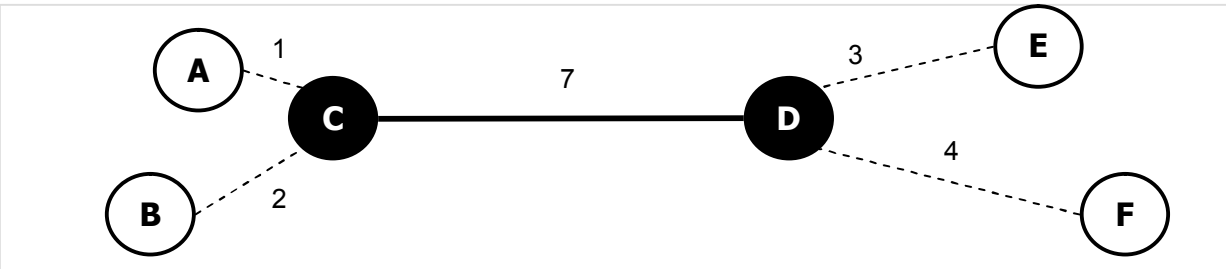


Рис. 28-4 — Паросочетание максимального веса (не наибольшее)

Итак, предстоящая задача формулируется следующим образом. Дан произвольный граф или орграф. В зависимости от заданного параметра поиска, надо найти в нём одно из паросочетаний: a) *наибольшее* без учёта веса рёбер или дуг; b) *наибольшее минимального* веса; c) *наибольшее максимального* веса. Если существует несколько решений одинаковой стоимости, алгоритм должен выдать любое из них. Паросочетания *наименьшего* и *наибольшего* веса будем называть *оптимальными*.

28.2. Основные идеи

Идея решения исходит из того, что ребро или дуга эквивалентны паре вершин, а пара — это подмножество вершин графа. В паросочетании эти подмножества не должны взаимно пересекаться, стало быть, можно подбирать нужную комбинацию *перебором* подмножеств из пар вершин (или подмножеств из **M** вершин — в общем случае). В итоге всё сводится к операциям с множествами, но насколько это выполнимо? Ведь задача экспоненциально сложна! Подобными вещами мы занимались в 10-й главе в поиске оптимального разбиения множества: составляли целое из непересекающихся «кусочков». Здесь нам помогут те же приёмы: группировка рёбер в блоки и досрочный возврат в ходе их перебора.

Сначала сгруппируем все рёбра (или дуги) в блоки так, чтобы в каждом блоке оказались только рёбра, инцидентные одной из вершин, эту вершину назовём *меткой* блока. Тогда в ходе подбора из каждого блока будет отбираться лишь по одному ребру, что на много порядков снизит трудоёмкость. Чтобы приблизить момент получения оптимального результата, упорядочим рёбра внутри блоков и сами блоки некоторым образом. Так, для поиска паросочетания минимального веса (сведём всё к этому варианту) рёбра внутри блока отсортируем в порядке не убывания их веса. Блоки — множества подмножеств — упорядочим в порядке не убывания количества содержащихся в них рёбер с тем, чтобы мелкие блоки обрабатывались раньше. Подробнее о формировании блоков будет сказано ниже.

Найдя какое либо паросочетание, в той или иной степени близкое к искомому (текущее), мы будем затем отсекать заведомо тупиковые ветви перебора и досрочно возвращаться на предыдущий уровень рекурсии (блоки перебираем рекурсивно). Для этого надо быстро прогнозировать, может ли обработка последующих блоков улучшить текущий результат. Эту оценку надо делать исходя из мощности и веса текущего и прогнозируемого паросочетаний. Так, при поиске паросочетания *минимального* веса прогноз будет положительным, если соблюдается одно из условий:

- a) обработка последующих блоков может увеличить мощность паросочетания в сравнении с текущим лучшим;
- b) обработка последующих блоков может дать паросочетание той же мощности, что и текущее, но при этом меньшей стоимости.

Разумеется, что *положительный* прогноз будет заведомо оптимистичным, не гарантирующим улучшения паросочетания. Зато *отрицательный* приговор будет абсолютно надёжным.

Теперь рассмотрим способ быстрого получения такого прогноза. Напомню, что мы перебираем рёбра, сгруппированные в блоки с метками вершин *A*, *B*, *C* и т.д. Без потери общности будем считать, что блоки перебираются в алфавитном порядке. В ходе перебора мы накапливаем паросочетание и множество вершин, входящих в него. Стало быть, на момент прогноза нам известна мощность накопленного паросочетания, его стоимость, и множество входящих в него

вершин. То же самое известно о текущем оптимальном паросочетании (если оно уже получено). Пусть на момент прогноза после обработки блоков с метками *A* и *B* в накопитель включены рёбра *A-X* и *B-Y*. Тогда в ходе погружения в рекурсию блоки *X* и *Y* будут пропущены, а из остальных блоков, начиная с блока *C*, для прогноза можно взять по одному самому «дешёвому» ребру. Сложив их стоимость со стоимостью накопленного паросочетания, получим искомый прогноз. Остаётся сравнить его с текущим лучшим решением по критериям, указанным выше.

Итак, в предлагаемом алгоритме будут использованы следующие приёмы:

- группировка рёбер (дуг) в блоки так, чтобы каждое ребро или дуга входили лишь в один блок;
- сортировка рёбер (дуг) внутри блока в порядке не убывания веса;
- сортировка блоков в порядке не убывания размеров блоков;
- рекурсивный перебор рёбер в блоках;
- быстрое вычисление прогноза для отсечения заведомо бесперспективных ветвей перебора.

28.3. О формировании блоков

Блоки — множества — будут вмещать в себя все рёбра или дуги графа. В неориентированном графе ребро представлено двумя встречными линками одинаковой длины, но в блок будет включен один из них. В ориентированном графе две вершины тоже могут быть связаны двумя встречными дугами, в общем случае разной длины. Здесь в блок попадёт лишь одна из них: большей или меньшей длины — в зависимости от решаемой задачи.

Отметим, что время перебора зависит не только от общего количества линков, но и от распределения их по блокам: чем меньше блоков, тем быстрее перебор (меньше глубина рекурсии). Предпочтительней иметь дело с малым числом крупных блоков, чем с множеством мелких. Поскольку каждый линк может быть включён в один из двух блоков, распределение линков по ним зависит от порядка обработки вершин, рассмотрим пример на рис. 28-5.

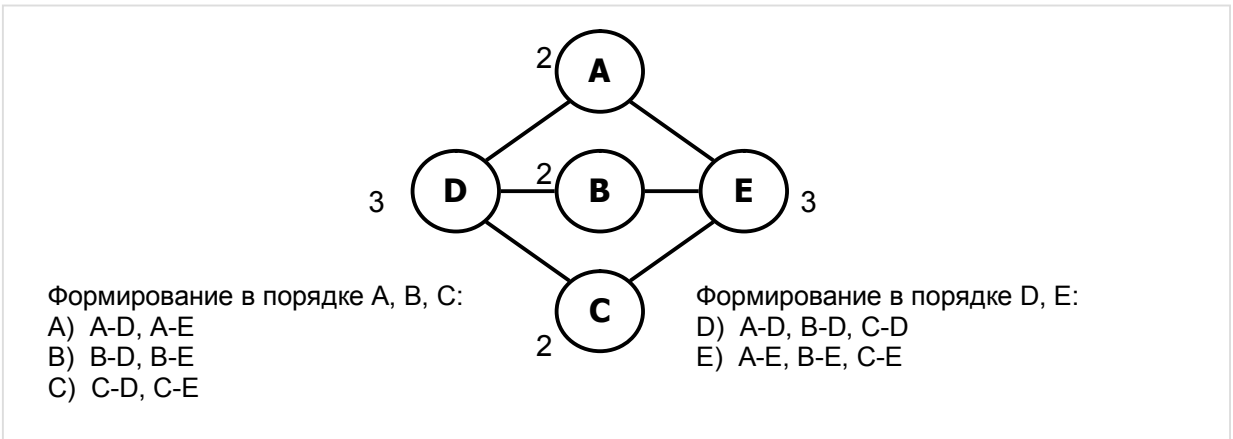


Рис. 28-5 — Состав и количество блоков зависят от порядка их формирования

Если в ходе формирования блоков обрабатывать вершины в алфавитном порядке, то будут порождены три блока по два линка в каждом (на рисунке слева). Если же формировать блоки в порядке *D*, затем *E*, то получим два блока по три линка (справа).

Для укрупнения блоков и уменьшения их количества поступим так. Перед формированием блоков отберём, как сказано выше, подходящие линки, окрасив их чёрным, а каждой вершине припишем число, равное количеству смежных ей чёрных линков (исходящих и входящих). Назовём это *степенью* вершины, начальные значения степеней показаны на рис. 28-5 рядом с вершинами. Затем выберем одну из вершин с максимальной степенью (вершину *D*) и сформируем блок с этой меткой, включив в него все линки, инцидентные данной вершине. По мере вставки линков в блок, степени инцидентных им вершин будем уменьшать всякий раз на единицу. И тогда мощность вершины *D* снизится с трёх до нуля, а степени трёх соседних с нею вершин (*A*, *B* и *C*) — с двух до единицы. Затем опять выберем вершину с максимальной текущей степенью (вершину *E*) и повторим с нею те же действия, в результате её степень и степени соседних вершин снизятся до нуля. В конце концов, формирование блоков остановится, когда все линки будут распределены, а степени всех вершин обнулятся.

Таким образом, выбирая всякий раз для формирования блока вершину с наибольшей текущей степенью и попутно корректируя эту степень в соседних вершинах, мы получим максимально крупные блоки, и минимальное их количество. Обсудив основные идеи, обратимся теперь к деталям алгоритма, и вначале опишем вспомогательные объекты: подмножество вершин (пару), и блок.

28.4. Вспомогательные объекты

Для представления пары вершин (линка) построим тип на базе класса **TCostSet**, который уже содержит в себе поле подмножества **mSet** и его стоимость **mCost**. Представление пары множеством позволяет при необходимости легко видоизменить алгоритм с целью поиска в графе **M**-сочетаний. Метод сравнения обеспечивает сортировку пар внутри блока по неубыванию стоимости.

Листинг 28-1 — Объект, представляющий пару вершин (ребро, дугу)

```
type TPair = class (TCostSet)
       function Compare(arg: TItem): TCompare;  override;
       procedure Print(var aFile: TextFile); override;
     end;

function TPair.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin Result:= cmpEq; Exit; end;
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене сравнением множества
    Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали, сравниваем по цене (дешёвые - в начало)
    if mCost > (arg as TCostSet).mCost
      then Result:= cmpGreate
```

```
      else Result:= cmpLess
  end;
end;
```

Следующий объект представляет блок линков; он построен на базе множества, и содержит дополнительные поля: метку блока (вершину) и стоимость самого «дешёвого» элемента в множестве.

Листинг 28-2 — Блок пар (линков)

```
type  TPairsBlock = class (TSetList)
        mLabel : TNode;    // метка блока (центральная вершина)
        mCost  : integer;  // вес наиболее лёгкой пары в блоке
        function Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: Text); override;
        destructor Destroy;  override;
      end;

destructor TPairsBlock.Destroy;
begin
  ClrAndDestroy;  // очистка с уничтожением элементов
  Inherited;
end;

// Блоки сортируются в порядке не убывания стоимости
// самой "лёгкой" пары в блоке

function TPairsBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;
  if Self = arg then Result:= cmpEq;
  if mCost > (arg as TPairsBlock).mCost then Result:= cmpGreate
end;
```

28.5. Метод поиска паросочетаний

В следующем листинге дан метод поиска паросочетаний, перебирающий подмножества. Основными его элементами являются функция построения блоков **GenBlocks**, рекурсивная процедура перебора **BlockHandle**, и функция предсказания **Prediction**.

Листинг 28-3 — Метод поиска паросочетаний в произвольном графе

```
// Метод генерации паросочетаний (п/с) трёх видов:
// pMaxN -- максимальной мощности (без учёта веса рёбер)
// pMaxW -- максимального веса
// pMinW -- минимального веса
function TGraph.GenPairs(aMode: TPairs): TCostSet;

var Blocks: TSet;         // Блоки пар (рёбер)
    BestCost: integer;    // Текущая лучшая (наименьшая) сумма
    BestCnt: integer;     // Текущая лучшая (наибольшая) мощность п/с
    Pairs: TCostSet;      // Текущее паросочетание
    Nodes: TSet;          // Текущее множество присоединённых вершин
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация вершин и дуг
  procedure InitNodes;
```

```
  var Node: TNode;
      LD, LR : TLink; // прямой и обратный линки
  begin
    // Предварительная очистка поля мощности
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // здесь будут подсчитываться смежные линки
      Node:= NodeNext;
    end;
    // Перебор всех линков с целью отбора наименьших или наибольших
    // (в случае наличия встречных линков)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Красим вершину белым
        mColor:= CWhite;
        // Перебор прямых линков LD
        LD:= OutLinkFirst;
        while Assigned(LD) do begin
          // Находим LR -- встречный линк
          LR:= LD.mDest.GetLink(Node);
          if Assigned(LR) then begin
            // Встречный линк найден
            if (aMode=pMaxW) and (LR.mValue < LD.mValue) then begin
              // При поиске максимума берём больший
              LD.mColor:= CBlack; // будет обрабатываться
              LR.mColor:= CWhite; // НЕ будет обрабатываться
            end else begin
              // При поиске минимума берём меньший
              LR.mColor:= CBlack; // будет обрабатываться
              LD.mColor:= CWhite; // НЕ будет обрабатываться
            end;
          end else begin
            // встречный линк НЕ найден
            LD.mColor:= CBlack; // будет обрабатываться прямой линк
          end;
          // Подсчёт инцидентных линков в двух соседних вершинах
          Inc(LD.mOwner.mPower);
          Inc(LD.mDest.mPower);
          LD:= OutLinkNext;
        end;
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины, способной дать блок максимального размера
  function GetBestNode: TNode;
  var Node: TNode;
      Max : integer;
  begin
    Max:= 0; Result:= nil;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do if (mColor=CWhite)     // вершина ещё не обработана
                   and (mPower > Max) // и её текущая мощность больше
      then begin
        Result:= Node;  // очередная лучшая вершина
        Max:= mPower;   // текущее число смежных чёрных линков
      end;
      Node:= NodeNext;
    end;
```

```
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование всех возможных пар вершин (рёбер)
  // и группировка их в блоки

  function GenBlocks: TSet;
    // Создание пары и вставка её в блок:
    function MakePair(aLink: TLink): TPair;
    begin
      Result:= TPair.CreateEmpty;
      with aLink do begin
        // В множество пары вставляем смежные с линком вершины:
        Result.mSet.Insert(mOwner);
        Result.mSet.Insert(mDest);
        // Уменьшаем мощности соседних вершин:
        Dec(mOwner.mPower);
        Dec(mDest.mPower);
      end;
      // Цену (вес) пары формируем в зависимости от варианта п/с
      case aMode of
        pMaxN: Result.mCost:=  1;              // наибольшее
        pMinW: Result.mCost:=  aLink.mValue;   // минимального веса
        pMaxW: Result.mCost:= -aLink.mValue;   // максимального веса ***
      end;
    end;
  var Node: TNode;          // текущая вершина
      Link: TLink;          // текущий исходящий линк
      Pair: TPair;          // текущая пара
      Min: integer;         // наименьший вес пары в блоке
      Block: TPairsBlock;   // текущий блок

  begin  { GenBlocks }
    InitNodes;            // инициализация полей mPower
    Result:= CreateSet;   // множество блоков
    // Перебор вершин:
    Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
    while Assigned(Node) do begin
      Min:= MaxInt;     // здесь формируется вес самой лёгкой пары в блоке
      Block:=nil;       // текущий блок пока не создан

      // Перебор исходящих линков:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Если линк не обработан
        if Link.mColor = CBlack then begin
          Link.mColor:= CWhite;  // восстанавливаем цвет
          Pair:= MakePair(Link); // создаём пару
          // Запоминаем вес легчайшей пары в блоке
          if Min > Pair.mCost then Min:= Pair.mCost;
          // Если блок ещё не создан, то создаём:
          if not Assigned(Block) then Block:= TPairsBlock.Create;
          Block.Insert(Pair);  // вставляем пару в блок
        end;
        Link:= Node.OutLinkNext;  // следующий линк
      end; // while

      // Перебор входящих линков:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Если линк не обработан
        if Link.mColor = CBlack then begin
          Link.mColor:= CWhite;  // восстанавливаем цвет
```

```
        Pair:= MakePair(Link); // создаём пару
        // Запоминаем вес легчайшей пары в блоке
        if Min > Pair.mCost then Min:= Pair.mCost;
        // Если блок ещё не создан, то создаём:
        if not Assigned(Block) then Block:= TPairsBlock.Create;
        Block.Insert(Pair);  // вставляем пару в блок
      end;
      Link:= Node.InLinkNext;  // следующий линк
    end; // while
    // Если блок создан, вставляем в множество результата:
    if Assigned(Block) then begin
      Block.mLabel:= Node;   // метка блока
      Block.mCost:= Min;     // вес легчайшей пары в блоке
      Result.Insert(Block);  // вставка блока
    end;
    Node.mColor:= CBlack; // отмечаем, что вершина обработана
    Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Рекурсивная процедура обработки блока

procedure BlockHandle(aBlock: integer;  // номер блока
                      aNodes: TSet;     // накопленные вершины
                      aRes: TCostSet    // накопленные пары
                      );
var Block: TPairsBlock; // текущий блок
    Nodes: TSet;        // Накопленное множество вершин
    Res  : TCostSet;    // Накопленное множество пар (рёбер)
    Pair : TPair;       // очередная пара из блока
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Прогноз результата

    function Prediction(aBlock: TPairsBlock;  // текущий блок
                        aNodes: TSet;         // накопленные вершины
                        aRes: TCostSet        // накопленные пары
                        ): boolean;
    var B: TPairsBlock;         // очередной блок
        Cost, Count : integer;  // прогнозируемые цена и мощность п/с
    begin
      Result:= not Assigned(aRes);
      if Result then Exit;
      // Начинаем накопление с текущей стоимости и мощности
      Cost:= aRes.mCost;  Count:= 0;
      if Assigned(aRes.mSet) then Count:= aRes.mSet.GetCount;
      // Пропускаем блоки вплоть до текущего
      B:= Blocks.GetFirst as TPairsBlock;
      while B <> aBlock do B:= Blocks.GetNext as TPairsBlock;
      // Переход к следующему блоку после текущего
      B:= Blocks.GetNext as TPairsBlock;
      // Перебор оставшихся блоков
      while Assigned(B) do begin
        if not aNodes.Exist(B.mLabel) then begin
          Inc(Count);         // накопление мощности паросочетания
          Inc(Cost, B.mCost); // накопление прогнозируемой стоимости
        end;
        B:= Blocks.GetNext as TPairsBlock;
      end;
      // Формируем положительный результат,
      // если выполняется одно из условий:
      // - есть шанс увеличить мощность паросочетания, или
      // - при той же мощности уменьшить стоимость паросочетания
```

```
        Result:= (Count > BestCnt) or
                 (Count = BestCnt) and (Cost < BestCost);
      end;
  begin { BlockHandle }
    // Извлекаем текущий блок:
    Block:= Blocks.GetItem(aBlock) as TPairsBlock;
    // Если метка текущего блока НЕ содержится в накопителе вершин,
    // то перебираем пары текущего блока
    if not aNodes.Exist(Block.mLabel) then begin
      // Вершина, которою помечен блок, ещё не входит в паросочетание
      // Перебор пар текущего блока:
      Pair:= Block.GetFirst as TPair;
      while Assigned(Pair) do begin
        // Попытка прилепить очередную пару (ребро)
        // Пара присоединяется, если множество пары
        // не пересекается с текущим множ. чёрных вершин
        if  not aNodes.TestIntersect(Pair.mSet) then begin
          // Здесь пару можно присоединить
          Nodes:= aNodes.Copy as TSet;      // копия накопленных вершин
          Res:= aRes.Copy as TCostSet;      // копия накопленных пар
          // Пристраиваем к множеству пар очередную пару
          Nodes.Add(Pair.mSet);    // накапливаем множество вершин
          Res.Insert(Pair);        // накапливаем множество пар
          if aBlock < Blocks.GetCount then begin
            // Это не последний блок.
            // Если обработка последующих блоков может увеличить
            // мощность паросочетания или снизить цену,
            // то войти в следующий блок
            if Prediction(Block,Nodes,Res)
              then BlockHandle(aBlock+1,Nodes,Res);
          end else begin
            // Достигнут последний блок (aBlock = Blocks.GetCount),
            // Запоминаем результат, если выполняется одно из условий:
            // - текущий результат Result ещё не определён
            // - мощность нового п/с превышает мощность текущего
            // - мощности п/с равны, но стоимость нового меньше
            if not Assigned(Result) or
               (Res.mSet.GetCount > BestCnt) or
               (Res.mSet.GetCount = BestCnt) and (BestCost > Res.mCost)
              then begin
                BestCost:= Res.mCost;          // лучшая (меньшая) стоимость
                BestCnt:= Res.mSet.GetCount;   // текущая лучшая мощность п/с
                Result.Free;                   // удаляем прежнее п/сочетание
                Result:= Res.Copy as TCostSet;// и сохраняем новое
              end;
          end;
          // Перед выбором следующей пары в блоке освобождаем текущие:
          Res.Free;   // накопленные пары
          Nodes.Free; // накопленные вершины
        end; // if
        Pair:= Block.GetNext as TPair;  // следующая пара в блоке
      end; // while
    end; // if
    // Если ищется наибольшее паросочетание без учёта веса,
    // и оно найдено, то выход
    if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
    // После перебора блока входим в следующий, если он существует
    if aBlock < Blocks.GetCount then begin
      // Текущий блок не последний.
      // Если обработка последующих блоков может увеличить
      // мощность паросочетания или снизить цену, то войти в следующий блок
      if Prediction(Block,aNodes,aRes)
```

```
        then BlockHandle(aBlock+1,aNodes,aRes);
    end else begin
      // Достигнут последний блок (aBlock = Blocks.GetCount),
      // Запоминаем результат, если выполняется одно из условий:
      // - текущий результат Result ещё не определён
      // - мощность нового п/с превышает мощность текущего
      // - мощности п/с равны, но стоимость нового меньше
      if not Assigned(Result) or
         (aRes.mSet.GetCount > BestCnt) or
         (aRes.mSet.GetCount = BestCnt) and (BestCost > aRes.mCost)
        then begin
          BestCost:= aRes.mCost;          // лучшая (меньшая) стоимость
          BestCnt:= aRes.mSet.GetCount;   // текущая лучшая мощность п/с
          Result.Free;                    // удаляем прежнее паросочетание
          Result:= aRes.Copy as TCostSet;// и сохраняем новое
        end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инверсия знаков весов рёбер паросочетания
  // (используется при поиске паросочетания максимального веса)

  procedure RestoreCost;
  var Pair : TPair;
  begin
    // Перебираем множество пар паросочетания Result.mSet
    with Result.mSet do begin
      Pair:= GetFirst as TPair;
      while Assigned(Pair) do begin
        Pair.mCost:= -Pair.mCost;
        Pair:= GetNext as TPair;
      end;
    end;
    Result.mCost:= -Result.mCost;  // итоговая стоимость всех пар п/с
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Перед уничтожением блоков удаляем из них найденные паросочетания

  procedure RemoveFromBlocks;
  var  Block: TPairsBlock;
  begin
    with Blocks do begin
      Block:= GetFirst as TPairsBlock;
      while Assigned(Block) do begin
        Block.Sub(Result.mSet);
        Block:= GetNext as TPairsBlock;
      end;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin
  Result:= nil;         // Результат -- лучшее паросочетание
  Blocks:= GenBlocks;   // Создаём блоки
  if Blocks.GetCount = 0 then begin Blocks.Free; Exit; end;

  // Подготовка переменных:
  BestCost:= MaxInt;             // текущая лучшая сумма
  BestCnt:= 0;                   // текущая лучшая (наибольшая) мощность п/с
  Nodes:= CreateSet;             // множество вершин паросочетания
  Pairs:= TCostSet.CreateEmpty; // начальное паросочетание
```

```
  // Рекурсивная обработка блоков:
  //Start:= Now;     // Время входа в процедуру
  BlockHandle(1, Nodes, Pairs);  // формирует Result

  // Очистка памяти:
  Pairs.Free;         // Начальное паросочетание (пустое)
  Nodes.Free;         // Начальное множество присоединённых вершин
  // Удаляем из блоков множество Result.mSet
  if Assigned(Result) then  RemoveFromBlocks;
  Blocks.ClrAndDestroy;  // Удаляем сами блоки
  Blocks.Free;           // и множество блоков

  // После поиска паросочетания максимальной стоимости
  // восстанавливаем стоимости линков:
  if aMode = pMaxW then RestoreCost;
end;
```

Остановим здесь внимание ещё на нескольких моментах.

Перед созданием блоков все линки, которые будут включаться в них, красятся чёрным, а после включения линка в блок, окраска снимается, что исключает повторное использование линка при обработке другой инцидентной ему вершины.

При формировании веса пары учитывается параметр, задающий тип искомого паросочетания: для паросочетания минимального веса вес пары не меняется, для максимального веса — инвертируется знак веса, а для поиска без учёта веса он принимается равным единице.

Блоки перебираются рекурсивной процедурой **`BlockHandle`**. Здесь при поиске паросочетания без учёта веса рёбер использовано ещё одно условие досрочного выхода:

```
    // Если ищется наибольшее паросочетание без учёта веса,
    // и оно найдено, то выход
    if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
```

Наконец, по завершении поиска, те пары, что попали в множество **`Result`**, надо удалить из блоков, иначе они будут уничтожены вместе с этими блоками при очистке памяти:

```
  if Assigned(Result) then  RemoveFromBlocks;
```

28.6. Испытания

В следующем листинге дана тестирующая программа, находящая в графе паросочетания трёх видов: без учёта веса рёбер, минимального веса, и максимального веса.

Листинг 28-4 — Программа для поиска паросочетаний трёх видов

```
{$APPTYPE CONSOLE}
```

```
uses
  SysUtils, DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TCostSet;
    Nodes: integer;

procedure Handle(arg: TPairs);
var Start: TDateTime;
    Time : integer;
begin
  Start:= Now;
  Res:= Gr.GenPairs(arg);
  Time:= MilliSecondsBetween(Start, Now);
  Res.Expo;
  Writeln('Nodes    = ', Gr.Nodes);
  Writeln('Time (ms)=', Time:6);
  Res.mSet.ClrAndDestroy;
  Res.Free;
end;
begin
      Gr:= TGraphChars.Load('Test.txt');
      Writeln('--------------- MaxN -----------------');
      Handle(pMaxN);
      Writeln('--------------- MinW -----------------');
      Handle(pMinW);
      Writeln('--------------- MaxW -----------------');
      Handle(pMaxW);
end.
```

В табл. 28-1 даны паросочетания для следующего неориентированного графа:

```
Неориентированный граф из 15 вершин и 52 рёбер
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
15 - количество вершин
A B C D E F G H I J K L M N O
A -> B=11 I= 1 K=98 M=14
B -> A=11 E=63 L=50
C -> E=47 H=87 I=50 M=10
D -> K=30 M=28
E -> B=63 C=47 L=70 N=37
F -> H=82
G -> J=74
H -> C=87 F=82 N=57 O=88
I -> A= 1 C=50 K=21 O=18
J -> G=74 L=77 O=99
K -> A=98 D=30 I=21 M=31 O=34
L -> B=50 E=70 J=77 M=38 N=59
M -> A=14 C=10 D=28 K=31 L=38
N -> E=37 H=57 L=59
O -> H=88 I=18 J=99 K=34
```

Табл. 28-1 — Результаты поиска паросочетаний в графе из 15 вершин

Вид паросочетания	Вес паросочетания	Множество рёбер
Без учёта веса рёбер	**7 (мощность п/с)**	A-B = 1 C-E = 1 D-K = 1 F-H = 1 G-J = 1 I-O = 1 L-M = 1
Минимального веса	**262**	C-M = 10 A-B = 11 I-O = 18 D-K = 30 E-N = 37 G-J = 74 F-H = 82
Максимального веса	**479**	J-O = 99 A-K = 98 F-H = 82 B-E = 63 L-N = 59 C-I = 50 D-M = 28

При поиске паросочетания минимального веса сформированы 9 блоков, они показаны в табл. 28-2, а звёздочками отмечены рёбра, вошедшие в паросочетание.

Табл. 28-2 — Блоки, сформированные при поиске минимального паросочетания

№	Метка блока	Вес легчайшего ребра	Подмножество рёбер
1	**M**	**28**	D-M = 28
2	**J**	**74**	G-J = 74 *
3	**E**	**37**	E-N = 37 * B-E = 63
4	**O**	**18**	I-O = 18 * J-O = 99
5	**A**	**1**	A-I = 1 A-B = 11 * A-M = 14
6	**H**	**57**	H-N = 57 F-H = 82 * H-O = 88
7	**C**	**10**	C-M = 10 * C-E = 47 C-I = 50 C-H = 87
8	**K**	**21**	I-K = 21 D-K = 30 * K-M = 31 K-O = 34 A-K = 98
9	**L**	**38**	L-M = 38 B-L = 50 L-N = 59 E-L = 70 J-L = 77

Перебор — занятие весьма трудоёмкое, и потому важно нащупать границы применимости этого алгоритма. Следующая программа может определять временные характеристики на графах трёх типов: a) полных, b) двудольных (все вершины долей соединены), и c) случайных с плотностью связей 20%.

Листинг 28-5 — Программа для исследования временных характеристик

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
var
  Gr : TGraph;
  Res : TCostSet;    // результат (паросочетание)
  Start: TDateTime; // для засечки времени
  Time : integer;
  Nodes : integer;  // количество вершин
  Retry : integer;  // количество повторений
  i : integer;
  Sum1, Sum2 : integer;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод результатов

procedure ResultExpo(const aFile: String);
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('-------------------------------------------------');
  Writeln('Nodes=', Nodes:4, ' Retry=', Retry:4);
  Writeln('Max= ', Sum1/Retry:7:1,  ' ms');
  Writeln('Opt= ', Sum2/2/Retry:7:1,' ms');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Abs(Nodes)<3 then Break;
    Write('Retry= '); Readln(Retry);
    if Retry=0 then Break;
    Sum1:= 0; Sum2:= 0;
    for i:= 1 to Retry do begin
      // Полный граф:
      // Gr:= TGraphChars.GenFull((Nodes>0), 0, 99, Abs(Nodes));
      // Двудольный насыщенный граф:
      // Gr:= THugeGraph.GenDicoty(99, Nodes);
      // Неориентированный граф 20% плотности:
      Gr:= TGraphChars.GenRandom(false, 0, 99, Nodes, -20{%});
      Write('.');
      // Без учёта веса дуг:
```

```
      Start:= Now;
      Res:= Gr.GenPairs(pMaxN);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum1, Time);
      Res.Free;
      // Минимального веса
      Start:= Now;
      Res:= Gr.GenPairs(pMinW);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      // Максимального веса
      Start:= Now;
      Res:= Gr.GenPairs(pMaxW);
      Time:= MilliSecondsBetween(Start, Now);
      Inc(Sum2, Time);
      Res.Free;
      Gr.Free;
    end;
    Writeln(#7);   ResultExpo('');
    ResultExpo('Out_17d.txt');
  until false;
end.
```

Отметим, что время поиска оптимального паросочетания берётся как среднее от времени поиска минимального и максимального паросочетаний. В следующих ниже таблицах показаны результаты испытаний на 50 случайных графах разной размерности.

Табл. 28-3 — Среднее время поиска паросочетаний в полных графах

Количество вершин	Среднее время поиска, ms	
	Без учёта веса вершин	Оптимальное паросочетание
9	0,6	9,4
10	0,6	11,6
11	0,6	145,0
12	0,6	156,0
13	0,6	2641,0
14	0,6	2703,0

Табл. 28-4 — Среднее время поиска оптимальных паросочетаний в случайных графах с плотностью связей 20%

Количество вершин	Среднее время поиска, ms	
	В орграфе	В неориентированном графе
20	18,5	61,8
21	41,6	135,5
22	61,4	294,0
23	175,3	632,0
24	324,8	1117,0
25	771,2	3157,0

Результаты для разреженных графов показаны на следующем рисунке.

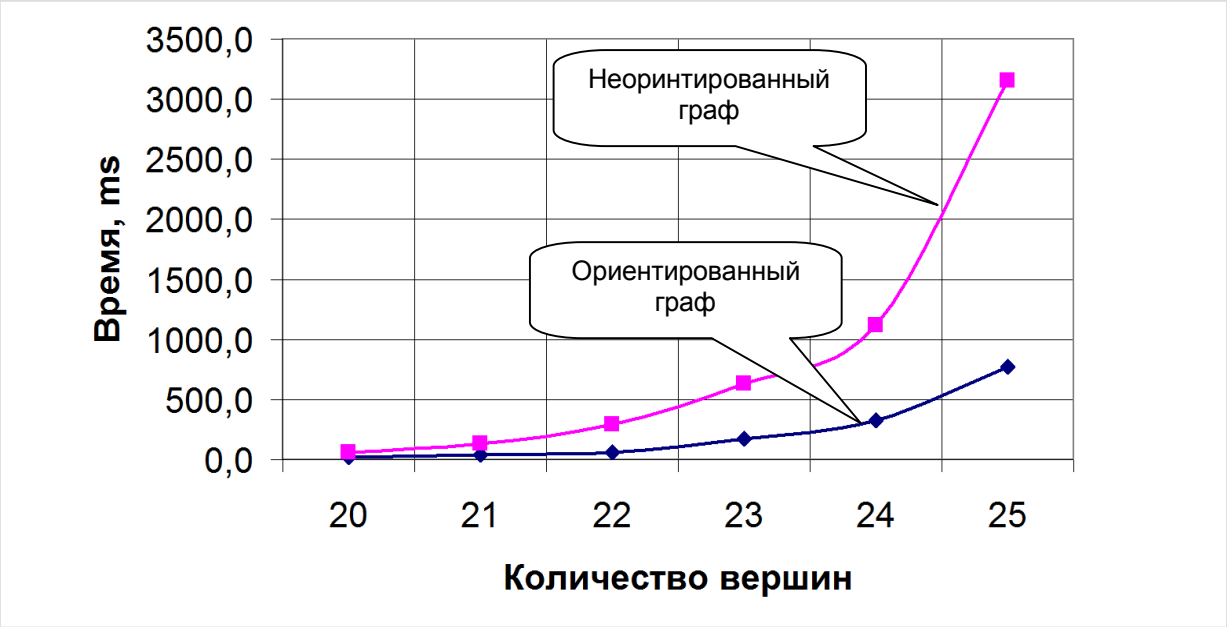


Рис. 28-6 — Среднее время поиска оптимальных паросочетаний в разреженных графах (плотность связей 20%)

Отсюда можно полагать, что граница применимости алгоритма будет тем шире, чем сильнее разрежен граф.

28.7. Итоги

28.7.1. Формирование пар из равноправных однородных объектов соответствует поиску паросочетаний в произвольном графе.

28.7.2. При формировании наибольших паросочетаний могут преследоваться следующие цели: получить *любое* паросочетание; получить паросочетание *наименьшей* или *наибольшей* стоимости (оптимальные). Все три типа задач можно свести к поиску паросочетания минимальной стоимости (веса).

28.7.3. Решение задачи поиска паросочетаний (и **м**-сочетаний) можно свести к подбору непересекающихся подмножеств.

28.7.4. Для снижения трудоёмкости перебора подмножеств используются следующие приёмы: формирование блоков из подмножеств (по возможности, крупных), сортировка подмножеств и блоков, перебор блоков с досрочным возвратом по результатам предсказания.

28.8. Задания

28.8.1. Ограничение попыток

В некоторых случаях расширить применимость перебора можно, пожертвовав точностью решения. Поскольку пары перебираются в порядке не убывания стоимости, алгоритму присуща некая похвальная жадность: даже первые промежуточные решения могут быть приемлемы. Реализуйте варианты алгоритма, ограничивающие либо количество промежуточных решений, либо общее время решения, либо время между получением промежуточных решений.

28.8.2. Поиск 3-сочетаний

Разработайте алгоритм для формирования оптимальных триад. Тройка вершин составит триаду, если одна из них — «бригадир» — соседствует с двумя другими.

28.9. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 405
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 178
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 29
Задача почтальона на неориентированном графе

Вот бедный китайский почтальон, а это — славный житель Кёнигсберга, великий математик Леонард Эйлер. Сейчас обнаружится связь между ними. Однажды у основателя теории графов во время прогулки по городу возник вопрос: можно ли пройти по всем его мостам и вернуться в исходную точку, преодолев мосты ровно по одному разу? Эйлер легко доказал, что для этого в графе, где мосты соответствуют рёбрам, все вершины должны иметь *чётную* степень. Действительно, если сжигать пройденные мосты, то после каждого посещения некоторой промежуточной вершины она будет лишаться двух рёбер. В конце концов, если у вершины останется одно ребро, мы, войдя в неё, не сможем затем покинуть. Стало быть, вершина с *нечётной* степенью не позволит решить задачу.

А что почтальон? Его задача несколько сложнее: разнести письма по всем улицам городка и вернуться в исходную точку, покрыв при этом минимальное расстояние. Если бы структура городка отражалась чётным эйлеровым графом (в котором все вершины имеют чётные степени), то задача решалась бы легко, а пройденное расстояние определялось бы суммой длин улиц. Но почтальону таких гарантий никто не дал: часть вершин имела нечётные степени, из-за чего некоторые улицы приходилось проходить повторно.

Сначала рассмотрим принцип решения задачи Эйлера — найдём цикл, в котором каждое ребро встречается единожды. Затем покажем, как сводится к циклу Эйлера задача почтальона.

29.1. Формирование цикла Эйлера

Итак, мы оперируем со *связным* неориентированным графом, все вершины которого имеют *чётные* степени (рис. 29-1). Далее для краткости такие вершины и граф будем называть *чётными*.

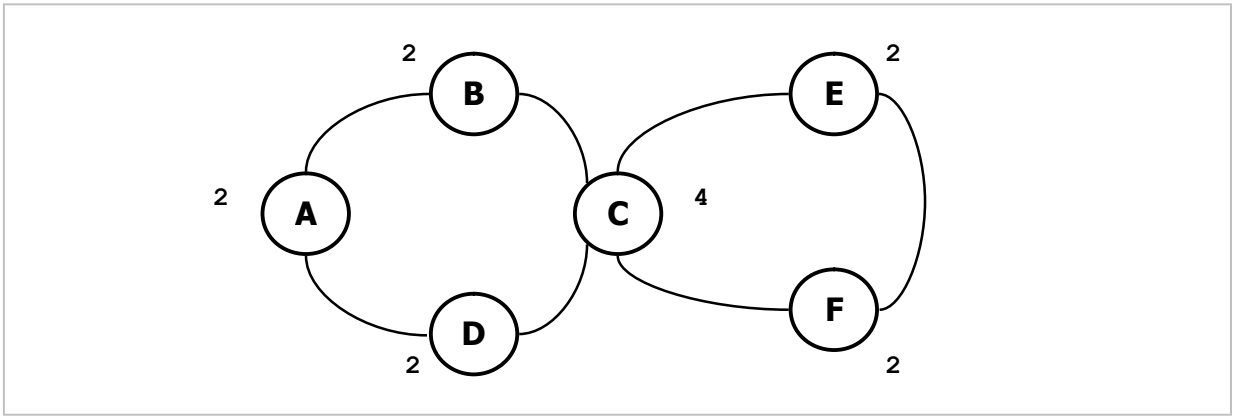


Рис. 29-1 — Связный неориентированный граф, все вершины которого имеют чётные степени

Легко догадаться, что результат не будет зависеть от стартовой вершины, поэтому для определённости начинать постройку цикла будем с первой вершины графа *A*. Припишем каждой вершине число, равное степени этой вершины, здесь у всех вершин, кроме *C*, степени равны двум. В ходе обхода рёбер будем уменьшать это число на единицу при каждом входе и выходе из вершины, Обход выполняем в глубину, двигаясь по ещё не пройденным рёбрам вплоть до исходной вершины.

Из вершины *A* можно попасть в *B*, а затем неминуемо в *C*. Здесь является альтернатива в виде вершин *D* или *E*. Предположим, мы двинулись в *E*, откуда затем неминуемо следуем в *F* и снова в *C*. Теперь альтернативы нет: в этой точке все «мосты», кроме *C-D*, «сожжены», и потому далее через вершину *D* возвращаемся в исходную вершину *A*. Посещены ли все «мосты»? Для ответа пройдём вдоль цикла и проверим вершинные числа: после посещений вершин и вычитаний все они обнулились, стало быть, задача решена, и нами получен цикл:

```
A-> B-> C-> E-> F-> C-> D-> A
```

Теперь рассмотрим случай, когда из вершины *C* мы двинулись сначала в *D*. Тогда мы достигнем исходной вершины *A* по короткому циклу:

```
A-> B-> C-> D-> A
```

Степени в этом цикле уменьшились на два, они обнулились у всех вершин, кроме *C*, а в вершине *C* степень уменьшилась с 4 до 2. Обходя цикл, мы обнаружим это, и тогда, начиная с вершины *C*, мы вновь построим цикл, двигаясь по ещё целым «мостам». Этот второй цикл надо вставить в разрез первого между вершинами **C->D**, и в итоге вновь получим тот же результат:

```
A-> B-> C->              D-> A    - 1-й цикл

           E-> F-> C->            - 2-й цикл

A-> B-> C-> E-> F-> C-> D-> A    - результат
```

Таким образом, последовательным построением циклов из стартовых вершин с ненулевыми степенями, и вставкой их в предыдущий цикл, достигается конечный результат: *эйлеров цикл*. Отметим, что вершины со степенями более двух при этом посещаются **P/2** раз, где **P** — степень вершины.

29.2. Приведение задачи почтальона к задаче Эйлера

Итак, вместе с Эйлером, нам удалось пройти по всем мостам по разу. Хорошо бы так же легко «щёлкнуть» и задачу почтальона, который, напомню, имеет дело с графом, где существуют *нечётные* вершины (*нечётным* графом в нашей терминологии). Напрашивается мысль сделать этот граф *чётным* добавлением рёбер к *нечётным* вершинам. Рассмотрим несколько простых закономерностей.

Пусть имеется чётный граф, все вершины которого чётны. Добавим к любой вершине ещё одно ребро, превратив её в нечётную. Это ребро неминуемо соединит вершину с другой чётной вершиной, и та тоже станет нечётной. В итоге нечётных вершин окажется две. Повторим это с другой чётной вершиной: теперь ребро соединит её либо с чётной, либо с нечётной вершиной. В первом случае нечётных вершин станет 4, а во втором их количество не изменится (останется 2). Отсюда следует вывод: *количество нечётных вершин в графе чётно*. Это наблюдение нам скоро пригодится.

Теперь на время превратимся в почтальона. Пусть в ходе постройки цикла Эйлера мы обошли все вершины графа некоторое число раз, уменьшая каждый раз степени вершин на два. В конце концов, степени нечётных вершин уменьшатся до единицы. Выйдя из некоторой нечётной вершины, и «сжёгши» последний инцидентный ей «мост», будет разумно последовать в другую нечётную вершину, поскольку количество *нечётных* вершин в графе *чётно*. Так, в конце концов, мы посетим все нечётные вершины (и чётные, разумеется, тоже). Стало быть, для сведения задачи почтальона к задаче Эйлера, надо всего лишь связать некоторые пары нечётных вершин дополнительными рёбрами, превратив их в чётные. Вопрос лишь в том, как внести при этом минимальное приращение к длине пути.

Здесь нам поможет алгоритм Флойда и Данцига, находящий кратчайшие пути между всеми вершинами графа, он обсуждался в главе 15. Пусть мы нашли все нечётные вершины, а также расстояния между ними. Из этого можно построить *полный* вспомогательный граф, где все нечётные вершины соединяются друг с другом рёбрами, длина которых равна длине кратчайшего пути между вершинами. Вы уже догадались, что паросочетание *минимального* веса в этом графе даст именно те пары нечётных вершин, которые надо соединить дополнительными рёбрами (о *паросочетаниях* см. в главе 28).

Так мы найдём минимальную длину пути, преодолеваемого почтальоном, а заменив дополнительные рёбра эквивалентными цепочками исходных рёбер, сформируем и сам путь (цикл). Осталось решить последнюю проблему: в наших графах нельзя соединять пару вершин более чем одним ребром, а это необходимо.

Осилим затруднение просто: припишем каждому ребру исходного графа число, которое назовём *степенью* ребра, и разрешим почтальону проходить по ребру столько раз, сколько указывает эта степень. Изначально все рёбра имеют единичную степень. Найдя оптимальные пути между нечётными вершинами в парах, мы нарастим степени рёбер вдоль этих путей, и тем некоторые рёбра удвоятся, утроятся, и т.д. В последующем, при построении циклов, каждое посещение ребра будет уменьшать его степень на единицу, так мы отследим доступные почтальону рёбра.

К слову сказать, после увеличения степеней рёбер, степени всех вершин графа станут *чётными*, если подсчитывать их с учётом степеней рёбер.

Теперь рассмотрим всю последовательность действий на примере небольшого графа, текст которого дан ниже:

```
Граф с четырьмя вершинами нечётной степени
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> C=1 D=6 E=10 F=8 G=10
B -> C=8 G=7 H=4
C -> A=1 B=8 D=6
D -> A=6 C=6
E -> A=10 F=4 H=6
F -> A=8 E=4
G -> A=10 B=7
H -> B=4 E=6
```

Граф показан на рис. 29-2, в нём есть 4 нечётные вершины, они выделены серым (степени вершин указаны вместе с их обозначением). Здесь нетрудно определить кратчайшие пути между нечётными вершинами (табл. 29-1).

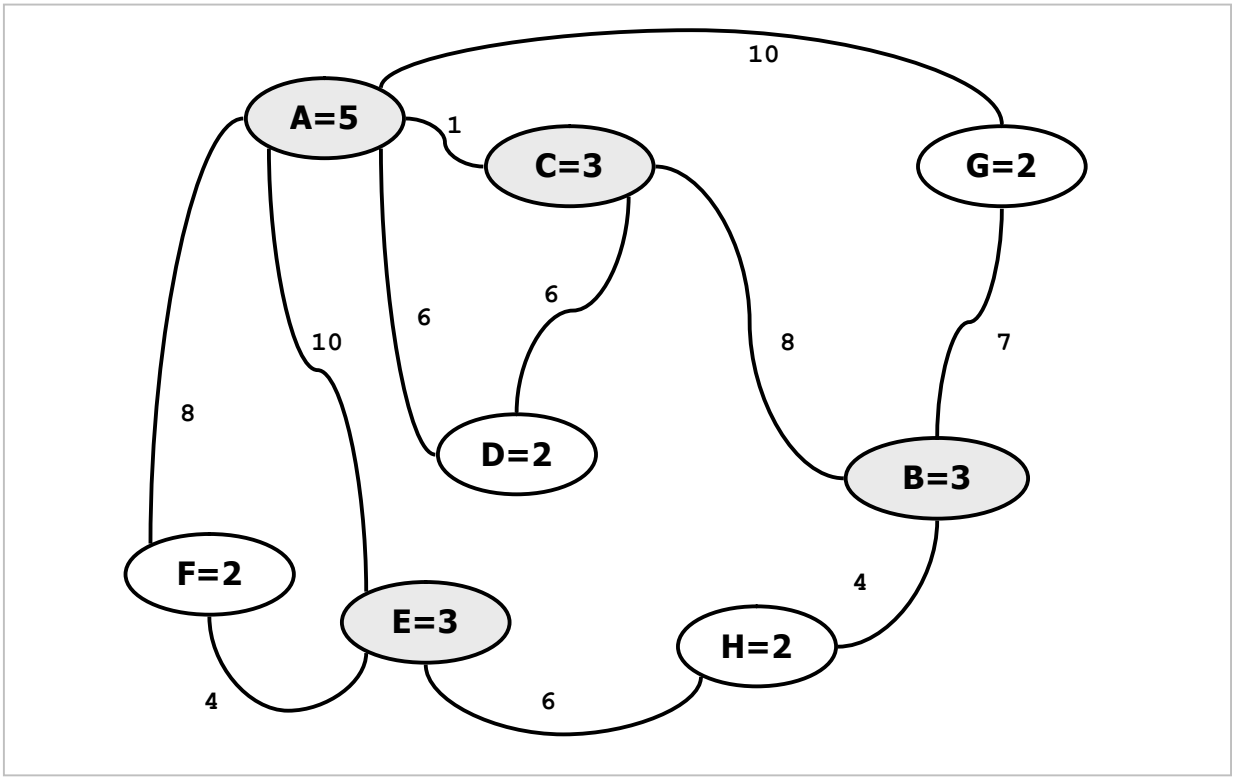


Рис. 29-2 — Взвешенный граф с четырьмя нечётными вершинами

Табл. 29-1 — Кратчайшие пути и расстояния между нечётными вершинами

Нечетные вершины	A	B	C	E
A	-	A-C-B = 9	A-C = 1	A-E = 10
B	A-C-B = 9	-	B-C = 8	B-H-E = 10
C	A-C = 1	B-C = 8	-	C-A-E = 11
E	A-E = 10	B-H-E = 10	C-A-E = 11	-

По этим данным строится вспомогательный полный граф и находится в нём паросочетание минимального веса (рис. 29-3).

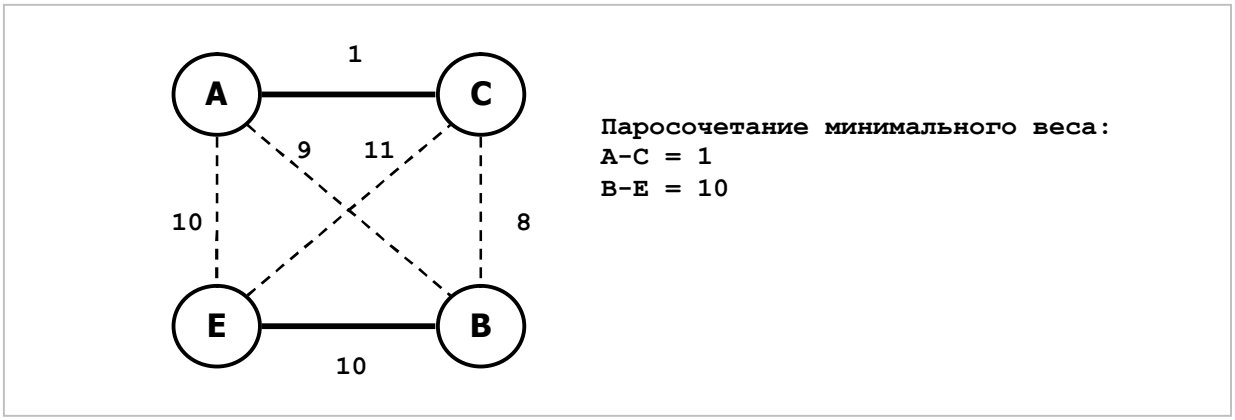


Рис. 29-3 — Вспомогательный полный граф и паросочетание минимального веса

Видно, что надо увеличить степени всех рёбер, лежащих на кратчайших путях между вершинами *A-C* и *B-E*, результат показан на рис. 29-4, где увеличены степени рёбер *A-C*, *B-H* и *H-E*. Теперь степени всех вершин тоже стали чётными, и граф готов для поиска эйлерова цикла, но уже с учётом степеней рёбер.

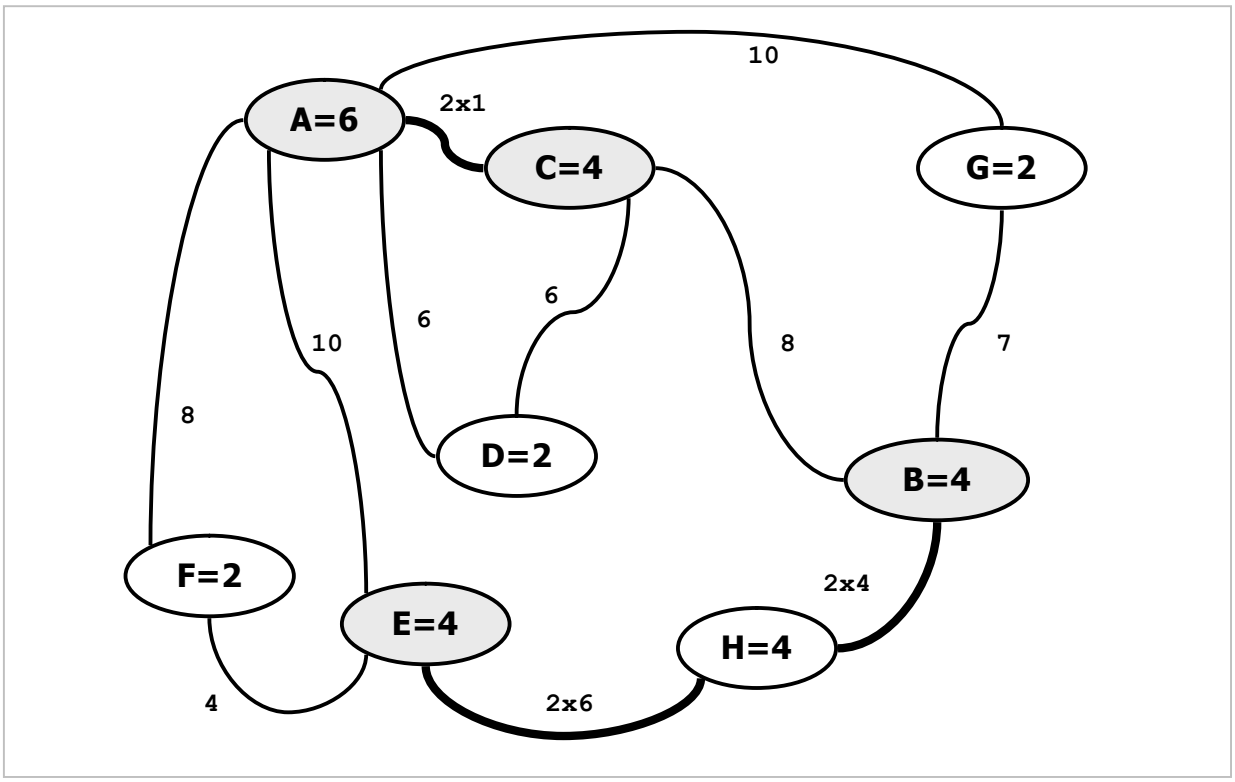


Рис. 29-4 — Граф после увеличения степеней некоторых вершин и рёбер

На рис. 29-5 показан порядок обхода графа из вершины *A*, тут числа рядом со стрелками обозначают последовательность обхода.

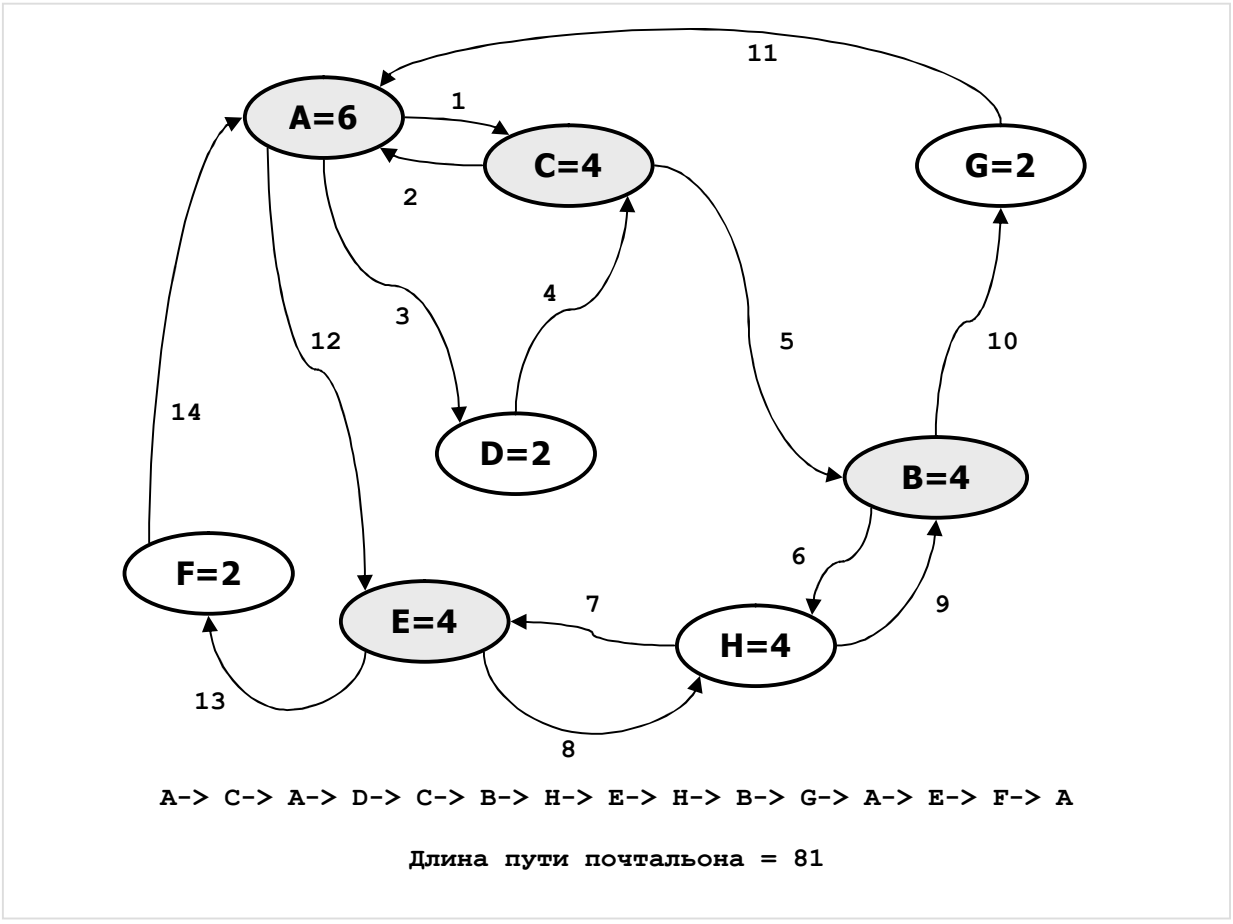


Рис. 29-5 — Порядок обхода рёбер из вершины A (числа указывают порядок)

29.3. Алгоритм

Дадим теперь алгоритм решения задачи почтальона:

- Подсчитать степени всех вершин графа и назначить рёбрам единичные степени.
- Если в графе существуют нечётные вершины, то:
  - Найти пути и расстояния между всеми нечётными вершинами.
  - Построить вспомогательный полный граф, состоящий только из нечётных вершин; найти в этом графе паросочетание минимального веса.
  - Нарастить степени рёбер исходного графа, лежащих на кратчайших путях между вершинами оптимальных пар, а также степени инцидентных им вершин.
- Выполнить обход графа в глубину, последовательно находя и объединяя циклы.

Всё это реализовано в методе **`GenPostPathUndir`**, текст которого дан ниже. Здесь для хранения текущих степеней вершин и рёбер отведены поля **`TNode.mLimit`** и **`TLink.mLimit`**.

Листинг 29-1 — Метод решения задачи почтальона на не ориентированном графе

```
function TGraph.GenPostPathUndir(var aCost: integer): TBuffer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Изменение степени ребра (двух полей TLink.mLimit)

  procedure ModifyLimit(aLink: TLink; aDelta: integer);
  var L: TLink; // встречный линк
  begin
    Inc(aLink.mLimit, aDelta);               // прямой линк
    L:= aLink.mDest.GetLink(aLink.mOwner);  // встречный линк
    Inc(L.mLimit, aDelta);
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества вершин с нечётной степенью
  // и предварительная подготовка полей TNode.mLimit

  function GenOddNodes: TSet;
  var Node: TNode;
  begin
    Result:= nil;
    // Перебор всех вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Сброс:  mColor= 0;  mPred= mLink= nil;  mDist= mFlow= MaxInt;
      Node.ResetNode;
      // В поле mLimit сохраним количество линков вершины (степень):
      Node.mLimit:= Node.OutGetCnt;  //Node.mLnkOut.GetCount;
      // Если количество рёбер нечётное, заносим в множество результата:
      if Odd(Node.mLimit) then begin
        if not Assigned(Result) then Result:= CreateSet;
        Result.Insert(Node);
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Установка степеней рёбер в полях TLink.mLimit
  // и степеней вершин в полях TNode.mLimit

  procedure LimitsPrepare;
  var OddNodes: TSet;     // множество вершин с нечётной степенью
        Pairs : TCostSet; // паросочетание минимального веса
         Pair : TPair;    // пара вершин
           Gr : TGraph;   // вспомогательный граф
      Ni, Nj  : TNode;    // вершины исходного графа
      N2i,N2j : TNode;    // копии вершин исходного графа во вспомогательном
         NFin : TNode;    // конечная вершина в кратчайшем пути
         Dist : integer;  // кратчайшее расстояние между вершинами
         Link : TLink;
         i, j : integer;

  begin
    // Устанавливаем степени всех рёбер в единицу:
    SetLinksLimit(1);
    // Собираем множество вершин с нечётными степенями:
    OddNodes:= GenOddNodes;
    // Если таких вершин нет, то выход:
```

```
    if not Assigned(OddNodes) then Exit;
    // Создаём вспомогательный граф:
    Gr:= TGraph.Create('Odd Nodes:', // имя графа произвольно
                       false,        // граф не ориентирован
                       false,        // вершины не нагружены
                       true          // рёбра нагружены
                      );
    // И вставляем в него копии вершин с нечётными степенями:
    Ni:= OddNodes.GetFirst as TNode;
    while Assigned(Ni) do begin
      Gr.InsertNode(Ni.Copy as TNode);
      Ni:= OddNodes.GetNext as TNode;
    end;
    // Для определения взаимных расстояний строим карту дальних указателей
    // в исходном графе:
    Self.InitMap_Floyd;
    // Строим рёбра между всеми вершинами вспомогательного графа
    // (создаём полный граф).
    // Длина рёбер устанавливается равной длине кратчайшего пути
    // между вершинами исходного графа
    for i:= 1 to OddNodes.GetCount-1 do begin
      Ni:= OddNodes.GetItem(i) as TNode;     // вершина из множества нечётных
      N2i:= Gr.mNodes.GetItem(i) as TNode;   // её копия в графе
      // Перебор последующих после Ni вершин:
      for j:= i+1 to OddNodes.GetCount do begin
        Nj:= OddNodes.GetItem(j) as TNode;   // вершина из множества нечётных
        N2j:= Gr.mNodes.GetItem(j) as TNode; // её копия во вспомогат. графе
        Dist:= Ni.GetFarLink(Nj).mDist; // кратчайшее расстояние в исх. графе
        Gr.SetLink(N2i, N2j, Dist);     // назначим длине ребра во вспомогат.
      end;
    end;
    // Во вспомогательном графе
    // находим паросочетание с минимальным весом:
    Pairs:= Gr.GenPairs(pMinW);

    // Для каждой из полученных пар увеличиваем степени mLimit
    // вдоль соответствующих кратчайших путей:
    Pair:= Pairs.mSet.GetFirst as TPair;
    while Assigned(Pair) do begin
      // Pair.mSet содержит ровно две вершины пары.
      // Стартовая вершина цепи в исходном графе:
      Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;
      // Конечная вершина цепи в исходном графе:
      NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
      repeat
        Inc(Ni.mLimit);          // степень+1 в начальной (промежут.) вершине
        Nj:= Ni.GetNear(NFin);   // следующая вершина на кратчайшем пути
        Link:= Ni.GetLink(Nj);   // линк на промежуточную вершину
        ModifyLimit(Link, +1);   // степень+1 линка
        Ni:= Nj;                 // продвижение к следующей вершине
      until Ni = NFin;           // пока не достигнута конечная
      Inc(Ni.mLimit);            // степень+1 в конечной вершине
      Pair:= Pairs.mSet.GetNext as TPair;  // следующая пара
    end;
    // Очистка памяти:
    Pairs.ClrAndDestroy;     // пары
    Pairs.Free;              // множество пар
    DoneMap;                 // освобождаем карту дальних связей
    Gr.Free;                 // вспомогательный граф
    OddNodes.Free;           // множество вершин с нечётной степенью
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  // Добавление кольцевых маршрутов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк
      // (степень которого Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл и путь:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (степень больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины пути)
      ModifyLimit(Link, -1);    // уменьшаем степень ребра
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;
    Link: TLink;
    Count: integer; // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathUndir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= 0;                // накопитель стоимости
  LimitsPrepare;            // подготовка полей mLimit (пределов посещ.)
  Node:= NodeFirst;         // построение начинаем с первой вершины

  // Строим кольцевые маршруты,
  // пока не исчерпаны степени всех вершин и рёбер
  repeat
    // Строим кольцевой маршрут из вершины Node и добавляем в результат:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;        // счётчик для прокрутки буфера Result
    while Count>0 do begin
      Link:= Result.Get as TLink;   // берём линк из начала буфера
      Result.Put(Link);             // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то стоп.
      // Построение очередного цикла начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход из цикла, когда степени всех вершин и рёбер исчерпаны:
  until not Assigned(Node);
end;

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
```

```
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;
```

Поясним некоторые технические моменты. Процедура **ModifyLimit** изменяет степень ребра, модифицируя поле **mLimit** сразу и в прямом, и в обратном линках ребра.

Процедура **LimitsPrepare** выполняет работу, предваряющую поиск циклов, она превращает нечётный граф (если он таков) в чётный. Здесь подсчитываются степени вершин, и, если надо, строится вспомогательный граф, находится минимальное паросочетание и корректируются степени рёбер. Отметим, что операторы:

```
      Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;
      NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
```

находят вершины в исходном графе, соответствующие их копиям во вспомогательном графе.

Процедура **AddPath** добавляет к буферу результата **Result** очередной цикл, начинающийся из вершины **aNode**. Объединение циклов происходит в главной программе, причём очередная часть цикла вставляется в нужное место путём прокрутки буфера **Result**.

29.4. Испытание

Следующую программу можно использовать для проверки метода, решающего задачу почтальона в неориентированном графе.

Листинг 29-2 — Программа для решения задачи почтальона на неориентированном графе

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
```

```
    Res : TBuffer;
    Cost: integer;

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Res:= Gr.GenPostPathUndir(Cost);
    Res.Expo;
    Res.Free;
    Writeln('Cost= ', Cost);
    Readln;
end.
```

На следующем рисунке представлен граф для проверки метода.

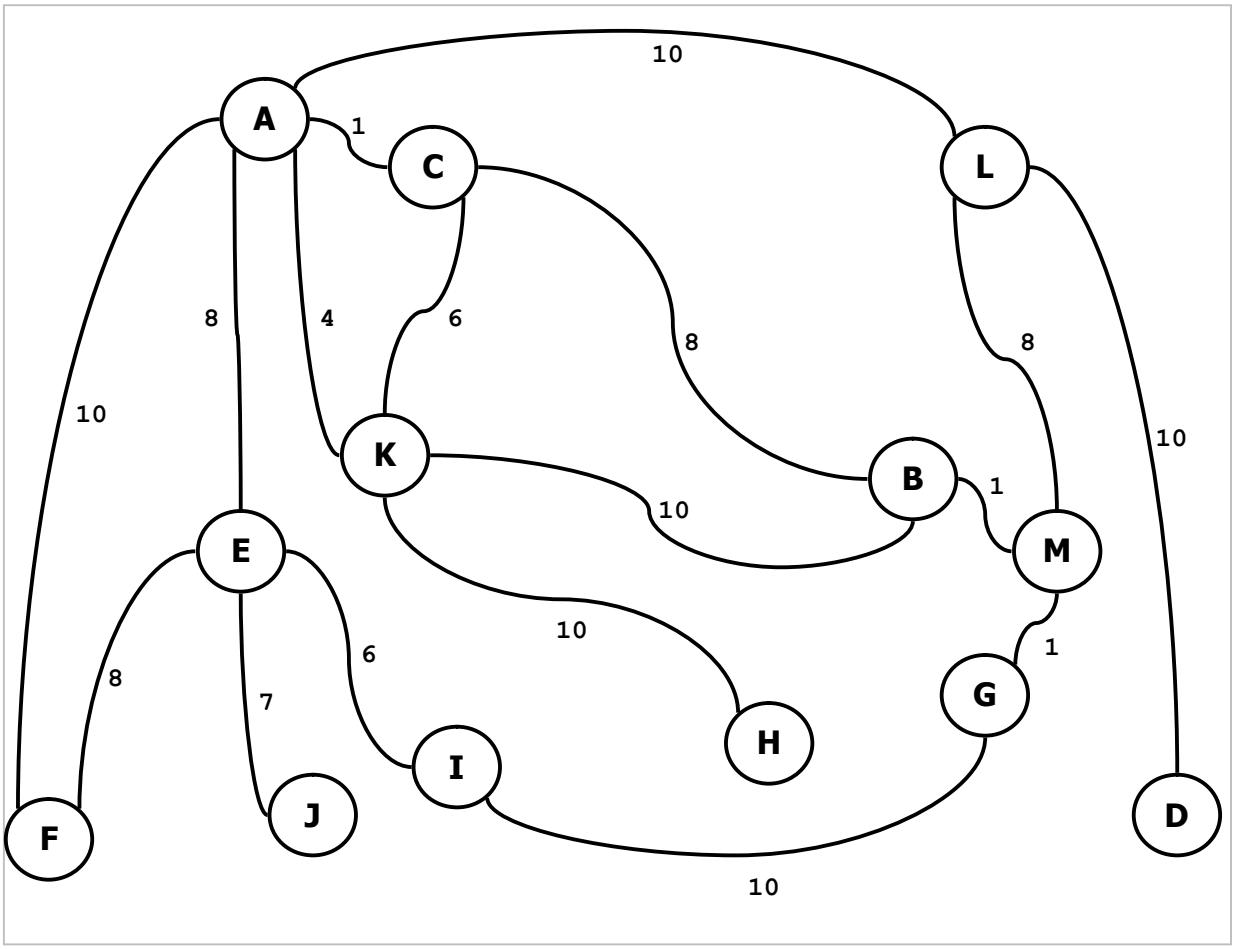


Рис. 29-6 — Граф для проверки метода, решающего задачу почтальона

Текстовое представление графа дано ниже:

```
Граф для проверки метода решения задачи почтальона
0 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
13 - количество вершин
A B C D E F G H I J K L M
A -> C=1 E=8 F=10 K=4 L=10
B -> C=8 K=10 M=1
C -> A=1 B=8 K=6
D -> L=10
E -> A=8 F=8 I=6 J=7
F -> A=10 E=8
G -> I=10 M=1
H -> K=10
I -> E=6 G=10
J -> E=7
K -> A=4 B=10 C=6 H=10
L -> A=10 D=10 M=8
M -> B=1 G=1 L=8
```

В результате получен цикл стоимостью 159 единиц, содержащий 25 рёбер:

```
A-> C-> A-> E-> J-> E-> A-> F-> E-> I-> G-> M-> B-> C-> K-> H-> K-> A-> K->
B-> M-> L-> D-> L-> A
```

29.5. Итоги

29.5.1. В задачах Эйлера и почтальона на неориентированных графах ищутся кратчайшие циклы обхода всех рёбер графа.

29.5.2. В задаче Эйлера надо пройти по всем рёбрам графа ровно один раз. Она имеет решение только для чётных связных графов, то есть таких, в которых все вершины имеют чётные степени.

29.5.3. Задача почтальона решается на произвольных связных графах, она состоит в поиске кратчайшего цикла, включающего все рёбра графа, при этом некоторые рёбра в цикле могут повторяться.

29.5.4. Решение задачи почтальона сводится к решению задачи Эйлера путём ввода в граф дополнительных рёбер (или повышения степеней существующих), и превращения тем самым нечётного графа в чётный.

29.5.5. Для поиска удваиваемых (утраиваемых, и т.д.) рёбер строится вспомогательный полный граф и отыскивается на нём паросочетание минимального веса.

29.6. Задания

29.7. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 227
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 219 Стр. 227
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 30
Задача почтальона на орграфе

Теперь расскажем о разбогатевшем почтальоне, — он уже не пеший, а развозит почту на автомобиле, и потому заботится об экономии топлива. Увы, улицы тамошнего городка слишком узки, и допускают лишь одностороннее движение. Почтальон ищет кольцевой маршрут (цикл) минимальной длины, проходящий по всем улицам городка хотя бы по разу. Понятно, что эта задача должна решаться на ориентированном графе.

Сначала выясним *необходимое* условие её решения. Очевидно, что граф должен быть *сильно связным*, то есть таким, что все вершины в нём взаимно достижимы. Будем считать, что оно соблюдается. Подробно связность графа обсуждалась в главе 16.

Второй вопрос: при каком условии все дуги будут пройдены ровно по одному разу? Это был бы самый подходящий для почтальона исход, он соответствует циклу Эйлера. Из того, что число входов в каждую вершину должно совпадать с числом выходов из неё, следует, что количество входящих в каждую вершину дуг должно совпадать с количеством исходящих (полу-степени входа и выхода должны совпадать). Графы, где это условие соблюдается, называются *симметричными*.

Но почтальону не повезло: граф, изображающий улицы и перекрёстки городка, оказался не симметричным (хотя и связным). Отсюда следует уточнение задачи: прежде, чем строить замкнутый маршрут почтальона (цикл), найдём множество дуг и вершин, которые ему придётся проезжать дважды, трижды, и т.д.

30.1. Идея решения

Введём подходящие термины. Назовём *степенью вершины* сумму количеств дуг, входящих и исходящих из вершины, то есть, сумму полустепеней исхода и захода. *Степенью дуги* назовём количество разрешённых посещений этой дуги, изначально степени всех дуг составляют единицу. Так же, как в задаче почтальона на неориентированном графе, в ходе поиска циклов степени вершин и дуг будут уменьшаться.

Назовём *асимметрией* вершины разность между количеством входящих и исходящих дуг. Если асимметрия больше нуля, значит, полустепень входа превышает полустепень исхода.

Теперь позволим себе мысленный эксперимент, и покажем его на следующем орграфе (рис. 30-1). Вообразим, что у каждой улицы городка есть свой почтальон, а в каждой вершине графа размещено почтовое отделение. Поутру почтальоны приходят каждый в *своё* отделение, садятся в служебные авто, обслуживают *свои* улицы, и оставляют автомобили уже в других почтовых отделениях (на другом конце улицы). В сумме все они делают ту же работу, что должен выполнить один,

но вот незадача: после работы в одних отделениях (вершинах) будут скапливаться лишние автомобили, а в других обнаружится их недостаток (надо ли объяснять, в каких?). Стало быть, по окончании смены надо перегнать автомобили из тех отделений, где они скопились, туда, где их недостаёт. Разумно было бы сделать это *кратчайшими* маршрутами. Читатель уже догадался, что эта задача равнозначна поиску *потока минимальной стоимости* из нескольких источников в несколько стоков, где единицами потока являются перегоняемые автомобили. Решение этой задачи даст оптимальное распределение потока по дугам, и этот поток отметит дуги, которые повторно посетит наш почтальон, работающий в одиночку.

Обратимся к рис. 30-1, здесь асимметрия чёрной вершины *A* составляет **4-1=3**, у серых вершин *B*, *C*, и *E* асимметрия равна **-1**, а прочие вершины симметричны.

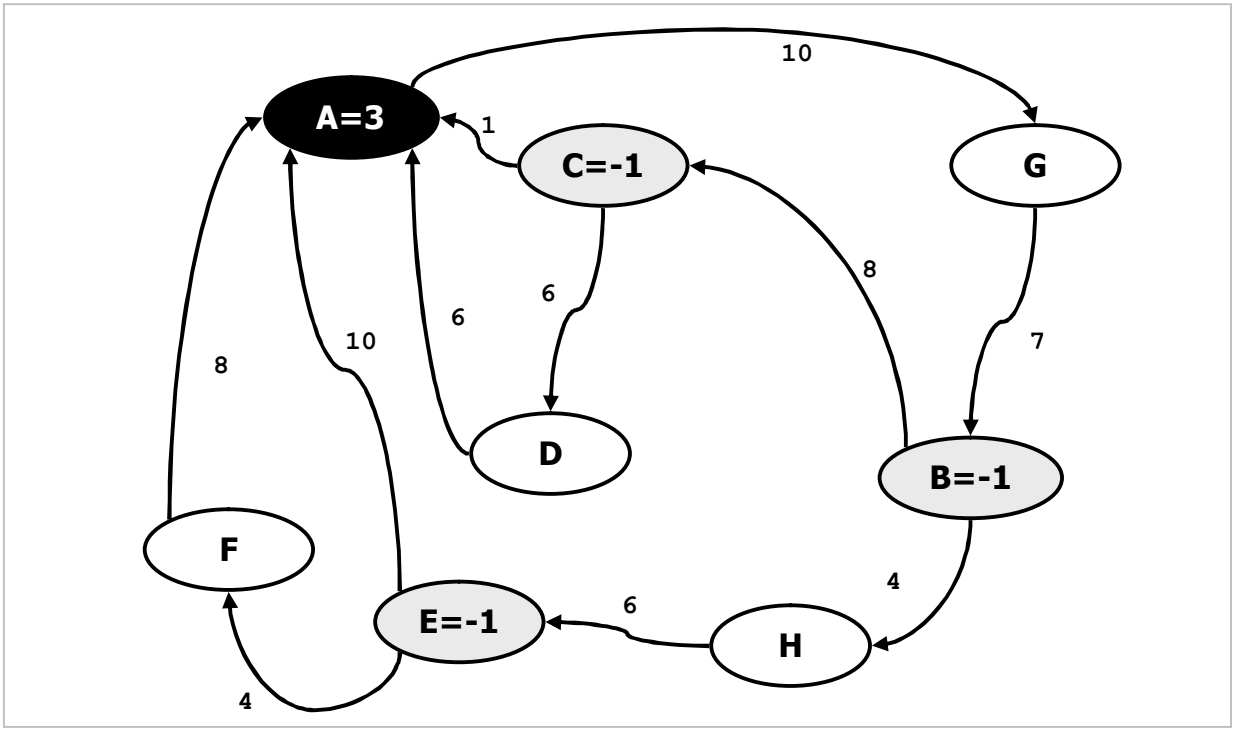


Рис. 30-1 — Орграф с четырьмя асимметричными вершинами

Понятно, что после почтовой смены в чёрной вершине окажется на три автомобиля больше, а в трёх серых будет недоставать по одному авто. Стало быть, надо переправить поток из чёрных вершин в серые, и общий объём потока составит здесь **3** единицы. Способ решения таких задач хорошо известен. Вводим в граф две вспомогательные вершины: общий исток *S* и общий сток *T*. Соединяем общий исток *S* дугами нулевой стоимости со всеми истоками в графе (чёрными вершинами). Этим дугам назначим пропускную способность, равную «силе извержения» соответствующего истока. Аналогично поступает со стоками, соединяя их с искусственным стоком *T*. Всем прочим дугам графа назначаем неограниченную пропускную способность, а цену (длину) сохраняем. Так получается граф, показанный на рис. 30-2.

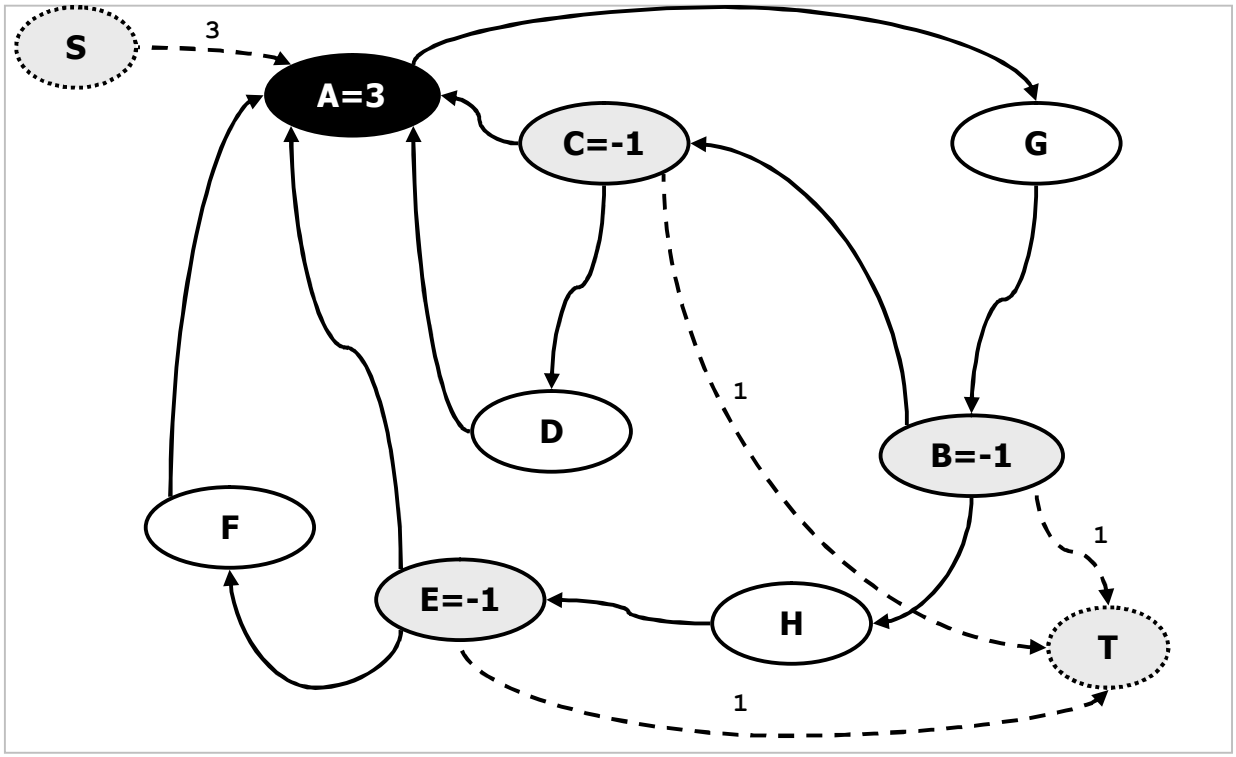


Рис. 30-2 — Дополнение орграфа истоком, стоком, и вспомогательными дугами

Распределение потока от S к T показано на рис. 30-3.

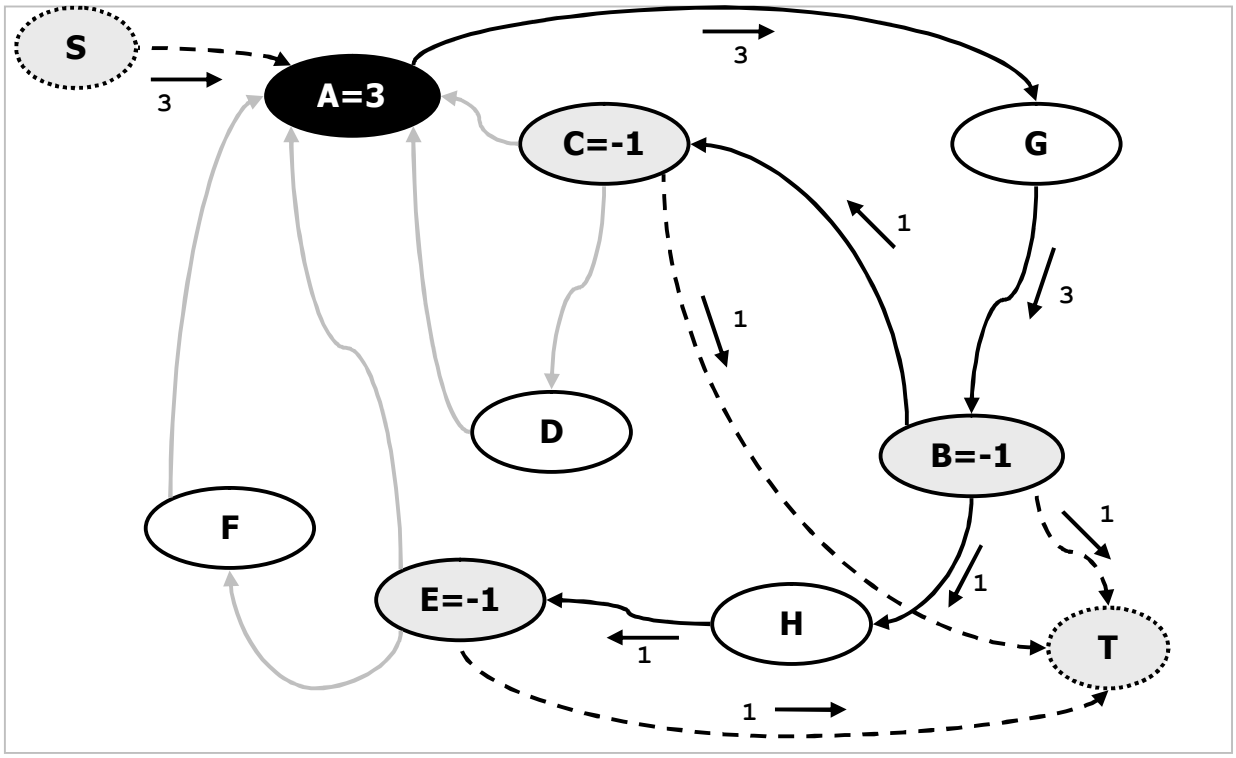
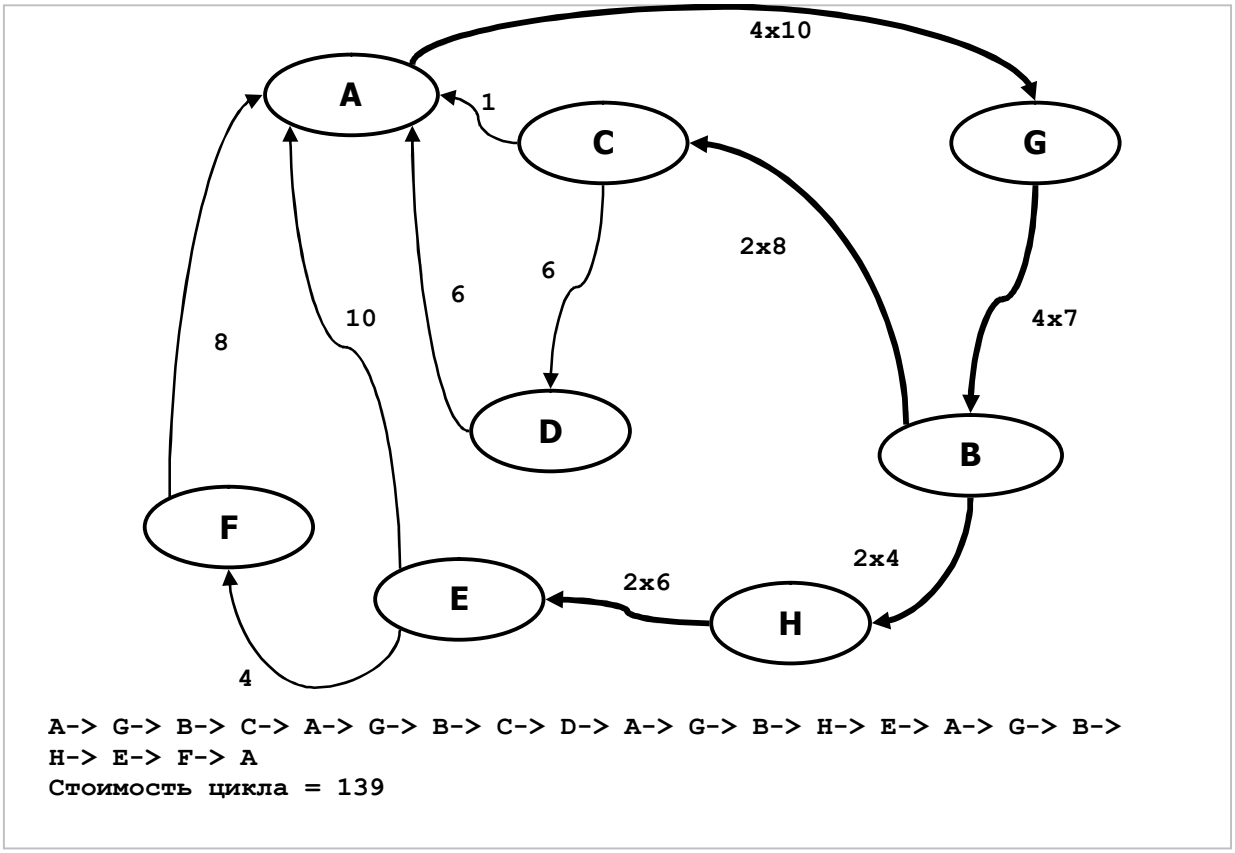


Рис. 30-3 — Распределение потока

Отсюда видим, что дуги A-G и G-B почтальону надо посетить ещё по три раза (всего по 4), а дуги B-C, B-H, H-E — дважды. Увеличив степени всех вершин с учётом найденного потока (и удалив вспомогательные дуги и вершины), получим граф, показанный на рис. 30-4.




```
A-> G-> B-> C-> A-> G-> B-> C-> D-> A-> G-> B-> H-> E-> A-> G-> B->
H-> E-> F-> A
Стоимость цикла = 139
```

Рис. 30-4 — Скорректированные степени дуг и кратчайший цикл

Далее на этом графе находим циклы Эйлера так же, как это делалось на неориентированном графе, результат показан на том же рисунке.

30.2. Алгоритм решения задачи почтальона на орграфе

* Вычислить начальные степени и асимметрию всех вершин. Назначить всем дугам единичные степени.
* Если граф оказался несимметричным, то:
    - Создать две вспомогательные вершины: искусственный исток S и сток T.
    - Соединить исток S со всеми положительно асимметричными вершинами, пропускную способность дуг назначить равной асимметрии этих вершин.
    - Соединить отрицательно асимметричные вершины с искусственным стоком T, пропускную способность дуг задать равной асимметрии этих вершин по модулю (положительной).
    - Найти распределение по дугам графа максимального потока минимальной стоимости от S к T.
    - Прибавить значения потоков в дугах к текущим степеням дуг и соответственно увеличить степени вершин.
* Найти кратчайший цикл Эйлера с учётом полученных значений степеней дуг и вершин.

Текст метода представлен в следующем листинге.

Листинг 30-1 — Метод для задачи почтальона на орграфе

```
function TGraph.GenPostPathDir(var aCost: integer): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подготовка полей mLimit в дугах и вершинах графа.
  // Возвращает стоимость дополнительного потока

  function LimitsPrepare: integer;
  var Node : TNode;     // текущая вершина
      Link : TLink;     // текущая дуга
      Flag : boolean;   // признак наличия асимметричных вершин
      S, T : TNode;     // искусственные исток и сток
      Flow: integer;    // величина корректирующего потока
  begin
    Result:= 0;
    // Устанавливаем пределы посещения всех дуг в единицу:
    SetLinksLimit(1);
    // Для всех вершин определяем разности полустепеней входа и выхода,
    // а также степени вершин TNode.mLimit:
    Flag:= false;         // признак асимметрии графа
    // Перебор вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mLimit:= InGetCnt + OutGetCnt;  // степень вершины
        mDist:= InGetCnt - OutGetCnt;   // асимметрия
        Flag:= Flag or (mDist<>0); // признак наличия асимметричных вершин
      end;
      Node:= NodeNext;
    end;
    // Если все вершины симметричны (все полустепени совпадают),
    // то выход из процедуры:

    if not Flag then Exit;

    // Для несимметричного графа строим поток минимальной стоимости
    // с величиной, равной суммарной ассиметрии
    // Создаём:
    S:= TNode.Create(0, Self); // вспомогательный исток
    T:= TNode.Create(0, Self); // вспомогательный сток

    Flow:= 0; // здесь подсчитаем суммарный поток
    // Обработка всех вершин, кроме S и T:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Во всех дугах формируем данные для вычисления потока:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mLow:= 0;            // минимальный поток = 0
        Link.mHigh:= MaxInt;      // максимальный не ограничен
        Link:= Node.OutLinkNext;
      end;
      // Если вершина несимметрична (mDist <> 0),
      // соединяем её с искусственным истоком либо стоком:
      if Node.mDist < 0 then begin
        // создаём дугу из вершины во вспомогательный сток:
        Link:= Node.MakeLink(T, 0); // создаём линк
        Link.mHigh:= -Node.mDist;   // пропускная способность дуги
      end else if Node.mDist > 0 then begin
        // Накапливаем требуемый поток:
        Inc(Flow, Node.mDist);      // асимметрия вершины
        // создаём дугу из вспомогательного истока в вершину:
```

```
        Link:= S.MakeLink(Node, 0); // создаём линк
        Link.mHigh:= Node.mDist;    // пропускная способность дуги
      end;
      Node:= NodeNext;
    end; // while
    // Вставляем вспомогательные вершины в граф:
    mNodes.Insert(S);          // исток
    mNodes.Insert(T);          // сток
    // Распределяем поток Flow по дугам и подсчитываем его стоимость:
    Result:= CalcMinCostFlow(S, T, Flow);
    // Удаляем из графа и освобождаем:
    RemoveNode(S);  S.Free;     // вспомогательный исток
    RemoveNode(T);  T.Free;     // вспомогательный сток

    // Добавляем полученные потоки в дугах к степеням дуг и вершин
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Flow:= Link.mFlow;  // поток в дуге
        // При наличии потока
        // добавляем его к степеням дуги и двух смежных вершин:
        if Flow > 0 then begin
          Inc(Link.mLimit, Flow);          // + к дуге
          Inc(Link.mOwner.mLimit, Flow);   // + к источнику дуги
          Inc(Link.mDest.mLimit, Flow);    // + к приёмнику дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление частичных циклов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк (Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (лимит больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины)
      Dec(Link.mLimit);         // уменьшаем степень дуги
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Node: TNode;      // текущая вершина
    Link: TLink;      // текущая дуга
    Count: integer;  // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathDir }
  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= -1;    // Стоимость цепи на случай отсутствия решения
```

```
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;

  aCost:= 0;           // накопитель стоимости
  LimitsPrepare;       // подготовка степеней mLimit вершин и рёбер
  Node:= NodeFirst;    // построение начинаем с любой вершины

  // Цикл добавления контуров к буферу Result:
  repeat
    // Строим циклы из текущей вершины Node
    // и добавляем в буфер Result:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;         // счётчик для прокрутки буфера
    while Count>0 do begin
      Link:= Result.Get as TLink;    // берём линк из начала буфера
      Result.Put(Link);              // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то
      // построение очередного контура начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход, если степени всех вершин исчерпаны
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);         // да, возвращаем назад в буфер
      Break;                     // и прекращаем цикл
    end;
    Result.Put(Link);            // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;
```

Этот метод сходен с методом для неориентированного графа: построение циклов выполняется одинаково. Отличие состоит лишь в предварительной подготовке степеней вершин и дуг.

30.3. Испытание

Для испытания может служить следующая программа.

Листинг 30-2 — Программа для решения задачи почтальона на ориентированном графе

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res : TBuffer;
    Cost: integer;

begin
    Gr:= TGraphChars.Load('Test.txt');
    Gr.Expo;
    Res:= Gr.GenPostPathDir(Cost);
    Res.Expo;
    Res.Free;
    Writeln('Cost= ', Cost);
    Readln;
end.
```

Проверим действие метода на графе, показанном на рис. 30-1, текстовое представление графа дано ниже:

```
Орграф для задачи почтальона
1 - тип графа (1 = орграф)
0 - вершины (1 = нагруженные)
1 - дуги (1 = нагруженные)
8 - количество вершин
A B C D E F G H
A -> G=10
B -> C=8 H=4
C -> A=1 D=6
D -> A=6
E -> A=10 F=4
F -> A=8
G -> B=7
H -> E=6
```

В результате получен следующий цикл длиной **139** единиц:

```
A-> G-> B-> C-> A-> G-> B-> C-> D-> A-> G-> B-> H-> E-> A-> G-> B-> H-> E->
F-> A
```

30.4. Итоги

30.4.1. В задачах Эйлера и почтальона на ориентированных графах ищутся кратчайшие циклы обхода всех дуг.

30.4.2. В задаче Эйлера надо пройти по всем дугам графа ровно один раз; она имеет решение только для симметричных сильно связных графов.

30.4.3. Задача почтальона решается на произвольных связных орграфах, она состоит в поиске кратчайшего цикла, включающего все дуги, при этом некоторые дуги в цикле могут повторяться.

30.4.4. Задача почтальона сводится к задаче Эйлера путём ввода в граф дополнительных дуг (или повышения степеней существующих дуг), — так несимметричный граф превращается в симметричный.

30.4.5. Для поиска удваиваемых (утраиваемых, и т.д.) дуг строится вспомогательный полный граф, затем в нём отыскивается максимальный поток минимальной стоимости.

30.5. Задания

30.6. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 227
8	Липский В.	Комбинаторика для программистов	
√ 9	Майника Э.	Алгоритмы оптимизации на сетях и графах	Стр. 219 Стр. 227
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 31
Разомкнутая задача Гамильтона

Задачи о посещении *всех* вершин графа отличает простота постановки и сложность решений. Но обо всём по порядку.

31.1. Контур и пути коммивояжёра

Отвлечёмся от текущих забот и вникнем в дело кочующего торговца — коммивояжёра. Этот почтенный господин периодически доставляет товары в ряд городов, связанных сетью дорог (не все города связаны напрямую). Источник его товара расположен в одном из городов, и потому торговцу надо избрать кратчайший маршрут, ведущий через *все* города и возвращающий его в исходную точку — такой маршрут назовём *контуром*. Эта задача известна как *замкнутый* вариант задачи о коммивояжёре.

Пусть теперь источники товара расположены в двух городах: *X* и *Y*. Тогда торговец будет следовать из *X* в *Y*, посещая по ходу *все* прочие города. Достигнув конечной точки, переночевав и запасшись товаром, он последует в обратном направлении от *Y* к *X*, заезжая опять в те же самые города. Поиск кратчайшего пути между двумя городами с посещением *всех* остальных городов известен как *разомкнутый* вариант задачи о коммивояжёре.

А где лучше расположить эти источники товара? То есть, выбрать города *X* и *Y* так, чтобы путь между ними через все прочие города оказался минимальным? — таково дополнение к *разомкнутому* варианту задачи о коммивояжёре.

31.2. Контур и пути Гамильтона

К сожалению, не все торговцы кристально честны, и некоторым лучше повторно не появляться там, где продан их товар. То же относится и к страховым агентам. Однако не всякая сеть наземных дорог позволяет им избежать нежелательных встреч: посещение *всех* городов может потребовать *повторного* визита в некоторые из них. И потому из гуманных соображений переместим место действия на острова, и свяжем острова морскими маршрутами так, чтобы из любого острова можно было добраться до любого другого, минуя прочие острова. Вы догадались, что так мы построили полный граф, однако заметим, что в этом графе прямое ребро или дуга между вершинами не всегда короче суммы нескольких промежуточных дуг (правило треугольника не гарантируем).

Итак, применительно к островам рассмотрим ещё три варианта задач о коммивояжёре, известных как задачи Гамильтона (Гамильтон — не торговец, а известный математик).

Контур Гамильтона — это замкнутый путь, проходящий через *все* вершины графа ровно по *одному* разу. Нас будет интересовать *кратчайший* контур.

Путь Гамильтона — это (кратчайший) путь, проходящий между двумя заданными вершинами с посещением всех остальных ровно по *одному* разу.

Дополнение к задаче о пути Гамильтона — поиск такой *пары вершин*, гамильтонов путь между которыми окажется кратчайшим для всех возможных пар.

Иногда в задачах коммивояжёра и Гамильтона ищут не *кратчайшие*, а *длиннейшие* пути. В таких случаях задачу сводят к поиску опять же кратчайших путей, заменяя длины дуг и рёбер на обратные. Далее мы всегда будем говорить о *кратчайших* путях.

31.3. Решаемость задач

Итак, задачи *коммивояжёра* отличает от задач *Гамильтона* только то, что коммивояжёру *разрешено* повторно посещать вершины, а Гамильтону — *нет*. Требования к контурам и путям коммивояжёра мягче, и потому легко догадаться, что задачи коммивояжёра всегда имеют решения на связном неориентированном графе или сильно связанном орграфе. Для задач Гамильтона критерии решаемости сложнее, но мы не станем глубоко входить в эту тему. Во-первых, потому, что нас будут интересовать главным образом полные графы, а на таких графах задачи Гамильтона всегда решаются. А во-вторых, в случае отсутствия решения на неполном графе, наши алгоритмы обнаружат это естественным путём.

На рис. 31-1 показан сильно связанный орграф, содержащий только контур коммивояжёра. Очевидно, что при обходе всех вершин графа здесь не миновать повторного посещения вершин *B* и *C*.

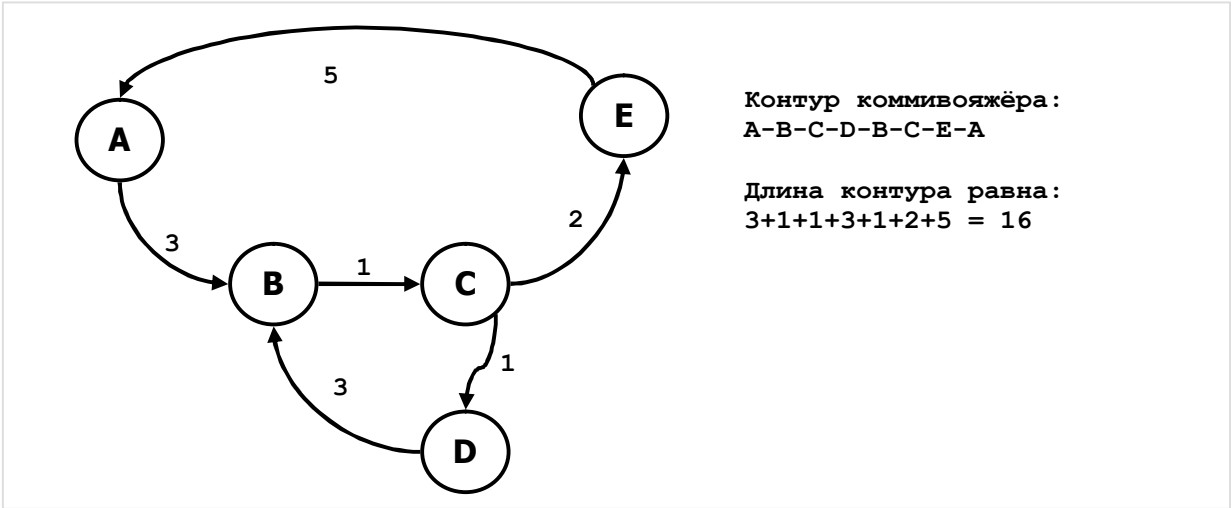


Рис. 31-1— Сильно связанный орграф, не содержащий контур Гамильтона

На рис. 31-2 показан почти тот же граф с добавленной дугой *D-E*, что позволило провести в нём ещё и контур Гамильтона (ни одна промежуточная вершина не посещена дважды). Обратите внимание, что здесь гамильтонов контур оказался длиннее кратчайшего контура коммивояжёра, и потому честному торговцу выгодней повторно посетить города *B* и *C*. Стало быть, если в графе

существует *гамильтонов* контур или путь, то существует и контур (путь) *коммивояжёра*, причём второй не длиннее первого.

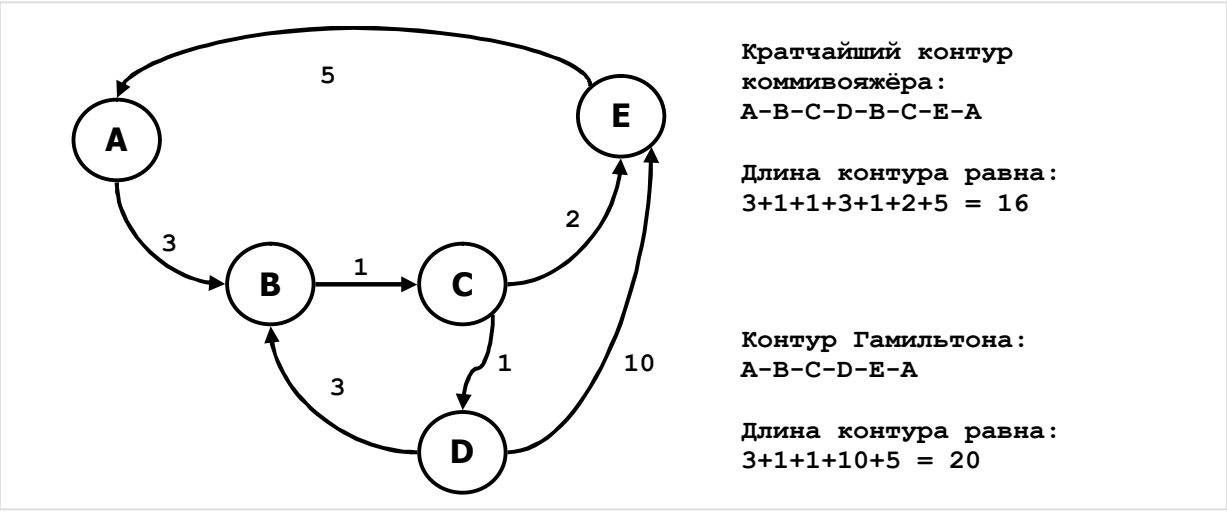


Рис. 31-2 — Орграф, содержащий и контур коммивояжёра, и контур Гамильтона

31.4. Связь между задачами *коммивояжёра* и *Гамильтона*

Математика щедра на парадоксы, и сейчас вас поджидает один из них: решать задачи *коммивояжёра* мы будем через... решение *гамильтоновых* задач!

Позвольте, но ведь сказано, что пути Гамильтона может не существовать даже при наличии пути коммивояжёра с его менее жёсткими требованиями! Фокус объясняется просто: *связанный* граф, на котором решается задача коммивояжёра, можно всегда свести к эквивалентному *полному* графу, соединив все его вершины виртуальными рёбрами или дугами. Каждой такой мнимой дуге или ребру можно сопоставить цепь из дуг или рёбер исходного графа, соединяющую пару вершин кратчайшим путём. Решив на таком графе задачу Гамильтона, решение для коммивояжёра получим путём обратной замены мнимых дуг (рёбер) цепочками настоящих дуг или рёбер.

Так, к примеру, отсутствующую дугу *D-E* на рис. 31-1 можно представить виртуальной дугой *D-E* такой, что длина её составит **3+1+2=6**, а это эквивалентно цепочке дуг *D-B-C-E*. Так связываются попарно все вершины графа, что порождает *полный* граф, на котором (гарантировано) решается гамильтонова задача.

Эта несложная техническая замена позволит нам впредь устремить все мысли на решение двух вариантов задачи *Гамильтона* : *открытой* (гамильтонов путь), и *закрытой* (гамильтонов контур). При внешнем сходстве формулировок, методы решения этих задач различны, и потому в этой главе мы уделим внимание только *открытой* задаче, а гамильтонов *контур* отложим до следующей главы.

Итак, основная открытая задача о кратчайшем *гамильтоновом пути* формулируется так: дан граф и две его вершины, необходимо найти *кратчайший*

путь из одной вершины в другую, проходящий через *все* прочие вершины графа ровно по одному разу.

31.5. Ориентиры

Забегая вперёд, отмечу, что в равной степени и *быстрого*, и *точного* решения этой задачи пока не найдено, и потому рассмотрим несколько алгоритмов, в разной степени отвечающим этим двум пожеланиям. Вначале обратимся к простым решениям, которые станут ориентирами, «маяками», в сравнении с которыми оценим эффективность других алгоритмов. Эти три простых алгоритма таковы: 1) случайный выбор, 2) жадный алгоритм и 3) полный перебор.

31.5.1. Случайный выбор маршрута

Этот абсурдный алгоритм обладает лишь одним достоинством — он очень быстр. Суть его соответствует названию: по пути от исходной вершины к конечной мы следуем «куда глаза глядят», переходя по первой попавшейся дуге в ещё не посещённую вершину (посещённые вершины окрашиваем). Ясно, что при испытании на большом количестве графов средняя длина маршрута будет близка к средней длине дуги, умноженной на количество дуг. Результаты, полученные случайным выбором, будут основой для оценки других алгоритмов. Реализацию случайного выбора мы объединим в одном методе с *жадным* алгоритмом.

31.5.2. Жадный алгоритм

В жадном алгоритме по пути от исходной к конечной вершине выбирается не произвольная, а *кратчайшая* дуга к ещё неокрашенной вершине. Посещённые вершины окрашиваются. Скорость алгоритма лишь немногим уступает случайному выбору, но результат существенно лучше (числовые сравнения даны в конце главы). Ниже следует листинг метода, в коем реализованы и случайный выбор, и жадный алгоритм, — вариант выбирается через параметр **aGreed**.

Листинг 31-1 — Случайный выбор и жадный алгоритм

```
function TGraph.GenHamPath_Greed(aStart, aFin: TNode;
                                 var aCost: integer;
                                 aGreed: Boolean // TRUE - жадный алгоритм
                                 ): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
          (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
```

```
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamPath_Greed }
  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                 // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;           // очистка вспомогательных полей
  aFin.mColor:= CBlack; // красим конечную вершину
  Cost:= 0;             // накопленная стоимость = 0

  Count:= mNodes.GetCount-2; // счётчик неокрашенных вершин
  Node:= aStart;
  while Count > 0 do begin
    Node.mColor:= CBlack;
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);    // накапливаем стоимость
    Dec(Count);             // счётчик - 1
    Node:= L.mDest;         // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины, кроме последней
    Result.Put(Node);       // заносим в результат предпоследнюю
    L:= Node.GetLink(aFin); // линк из предпоследней в последнюю
    if Assigned(L) then begin
      Result.Put(aFin);
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else Result.Clear;  // если нет завершающего линка
  end;
end;
```

31.5.3. Полный перебор маршрутов

Хотя жадному алгоритму присущи зачатки интеллекта, однако он, подобно начинающему шахматисту, видит лишь на шаг вперёд и лишён той прозорливости, что требует точное решение. А достичь такого результата на первый взгляд не так уж сложно: надо перебрать все возможные пути от исходной вершины к конечной. Рекурсивный алгоритм такого перебора представлен ниже, — это типичный обход графа в глубину, по ходу которого накапливается стоимость очередного пути. Возврат на предыдущий уровень рекурсии случается в двух ситуациях:

а) достигнута конечная вершина, и путь построен, — здесь запоминаем лучший на текущий момент результат;

б) накопленная стоимость пути превысила лучшую на данный момент, — тогда нет смысла углубляться, и потому отступаем на шаг.

Листинг 31-2 — Поиск минимального пути рекурсивным перебором

```
function TGraph.GenHamPath_Full(aStart, aFin: TNode;
                                var aCost: integer): TBuffer;
var
    Cost: integer;   // текущая стоимость контура
    Count: integer;  // счётчик окрашенных вершин
    Node: TNode;     // текущая вершина
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aFin - последняя окрашенная вершина
  procedure SaveResult;
  begin
    aCost:= Cost;          // сохраняем стоимость
    Result.Clear;          // очищаем буфер результата
    Node:= aFin;           // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=aStart;     // пока не достигнем начальной вершины
    Result.Push(Node);     // заносим конечную вершину
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);

  var L : TLink;  // текущий исходящий линк
  begin { Local }
    aNode.mColor:= CBlack;  // красим данную вершину
    Inc(Count);             // и наращиваем счётчик окрашенных
    // Все вершины (кроме последней) окрашены?
    if Count < mNodes.GetCount-1 then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
        if (L.mDest <> aFin) and (L.mDest.mColor = CWhite) then begin
          // Нашли неокрашенную и не последнюю:
          Inc(Cost, L.mValue);        // накапливаем стоимость
          if Cost < aCost then begin  // если стоимость ниже минимальной
```

```
            L.mDest.mPred:= aNode;     // то метим предыдущей вершиной
            Local(L.mDest);            // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);         // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;         // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount-1 -- все вершины, кроме aFin, окрашены
      // Ищем замыкающий линк на вершину aFin
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>aFin)
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);       // накапливаем стоимость
        L.mDest.mPred:= aNode;     // метим предыдущей вершиной
        if Cost < aCost            // если она меньше минимальной
          then SaveResult;         // сохраняем контур и его стоимость
        Dec(Cost, L.mValue);       // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenHamPath_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                  // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  aCost:= MaxInt;// текущая минимальная стоимость = Maxint
  ResetNodes;           // очистка вспомогательных полей
  Cost:= 0;             // накопленная стоимость = 0
  Count:= 0;            // счётчик окрашенных вершин = 0
  Local(aStart);        // вызов рекурсивной процедуры
  // Если путь не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;
```

Читатель, конечно же, подозревает, что простота и точность алгоритма чем-то оплачена. Чем? — временем, разумеется. Количество возможных путей из вершины *X* в вершину *Y* на полном графе с *N* вершинами составляет величину *(N-2)!*, и потому время полного перебора с ростом размера графа растёт с той же чудовищной факториальной крутизной. Нынешние компьютеры заметно «притормаживают» уже при *N* **= 15...20**, а к концу третьего десятка перед задачей капитулирует даже самый мощный из них (точнее, его пользователь).

Подведём промежуточный итог. У нас есть очень *быстрый* жадный алгоритм, находящий гамильтонов путь, пусть и далёкий от идеала, но гораздо короче случайного. С другой стороны, есть алгоритм полного перебора, дающий *точный* результат. К сожалению, из-за крайней медлительности полный перебор не применим к сколь-нибудь крупным графам. Отсюда видим нашу цель: найти

относительно *быстрый*, и по возможности *точный* алгоритм поиска минимального гамильтонова пути, — этому посвятим остаток главы.

31.6. Покрывающие деревья и штрафование вершин

Улучшенные методы решения гамильтоновых задач для *неориентированных* графов и *орграфов* принципиально отличаются. Создание метода для *орграфов* отложим до 33-й главы, а здесь рассмотрим построение кратчайшего гамильтонова пути (или почти кратчайшего) на *неориентированных* графах. Создаваемый метод отталкивается от построения минимального покрывающего дерева (остова) весьма быстрым алгоритмом *Прима*, который подробно описан в главе 23.

Но, при чём тут *Прим*? Дело в том, что искомый нами гамильтонов путь между вершинами *X* и *Y* является в то же время и одним из *покрывающих деревьев* графа. Пусть на некотором графе построен минимальный остов, и сложилось так удачно, что вершины *X* и *Y* в этом дереве имеют степени **1**, а все прочие вершины — степени **2** (степень вершины равна количеству инцидентных ей рёбер). Это значит, что образовалась цепочка рёбер от *X* к *Y*, — она и будет кратчайшим гамильтоновым путём. Конечно, столь удачное совпадение маловероятно, однако эту вероятность можно повысить, пустившись на невинный обман алгоритма Прима, известный как *метод штрафования вершин*.

Для упрощения дальнейших пояснений, введём в рамках этой главы несколько терминов. Назовём *n*-вершиной покрывающего дерева, вершину, степень которой в дереве равна *n*. Тогда **1**-вершины назовём *листьями*, **2**-вершины — *суставами*, а вершины степеней **3** и более — *узлами* (рис. 31-3).

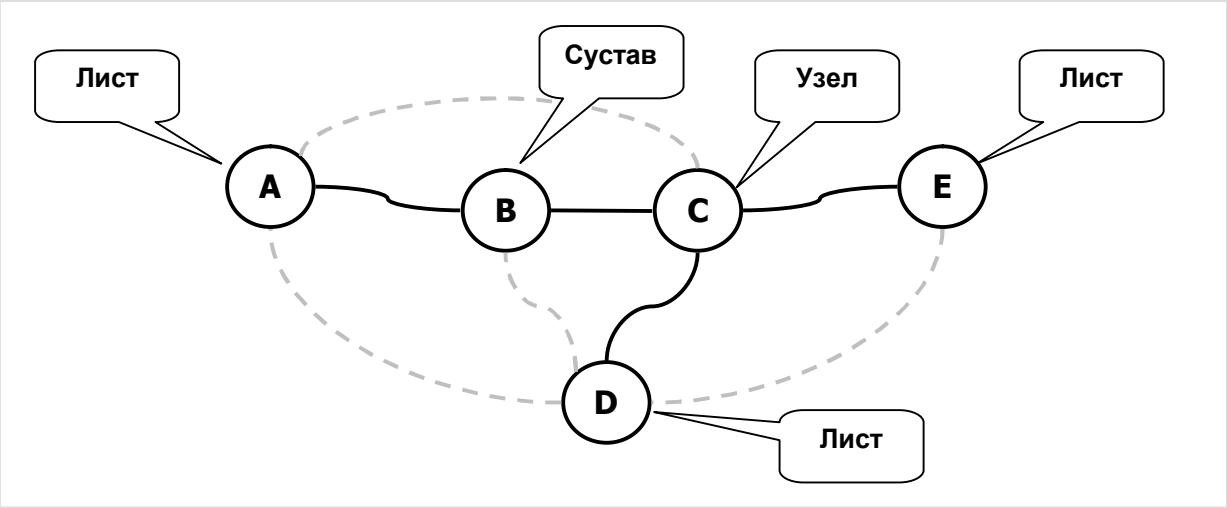


Рис. 31-3 — Остовное дерево с тремя листьями, одним суставом и одним узлом степени 3

В этих терминах нашу конечную цель можно выразить так: в искомом гамильтоновом пути вершины *X* и *Y* должны оказаться *листьями*, а все прочие вершины – *суставами*. Никаких *узлов* в этом дереве быть не должно. Однако

алгоритм Прима для кратчайшего покрывающего остова не даёт таких гарантий, и нам предстоит модифицировать его.

С этой целью обратим внимание на особенность алгоритма Прима: в ходе построения минимального остова вершины, плотно окружённые близлежащими соседями, «притягивают» их к себе, обрастая большим числом рёбер, и становясь *узлами* дерева. Наоборот, отдалённые от прочих вершины соединяются с деревом лишь одним ребром, превращаясь в *листья*. Поскольку общее количество рёбер в дереве фиксировано и равно $N-1$ (N — количество вершин), то количество *листьев* в дереве соответствует количеству и степеням его *узлов*. Таким образом, по количеству листьев L можно судить, насколько построенный остов отличается от пути Гамильтона. Разность $L-2$ — количество листьев за вычетом двух — назовём *невязкой*. Если невязка построенного остова окажется равной нулю, то этот остов будет *кратчайшим* гамильтоновым путём. Теперь обсудим пути снижения этой *невязки*.

Излагаемую далее технику называют *методом штрафования вершин*, хотя, что это объясняет? Чем провинились вершины? Мы дадим несколько иную — геометрическую — интерпретацию этого приёма. И пусть она не точна количественно, зато вполне наглядна. Рассмотрим граф, все вершины и рёбра которого лежат в одной плоскости (рис. 31-4).

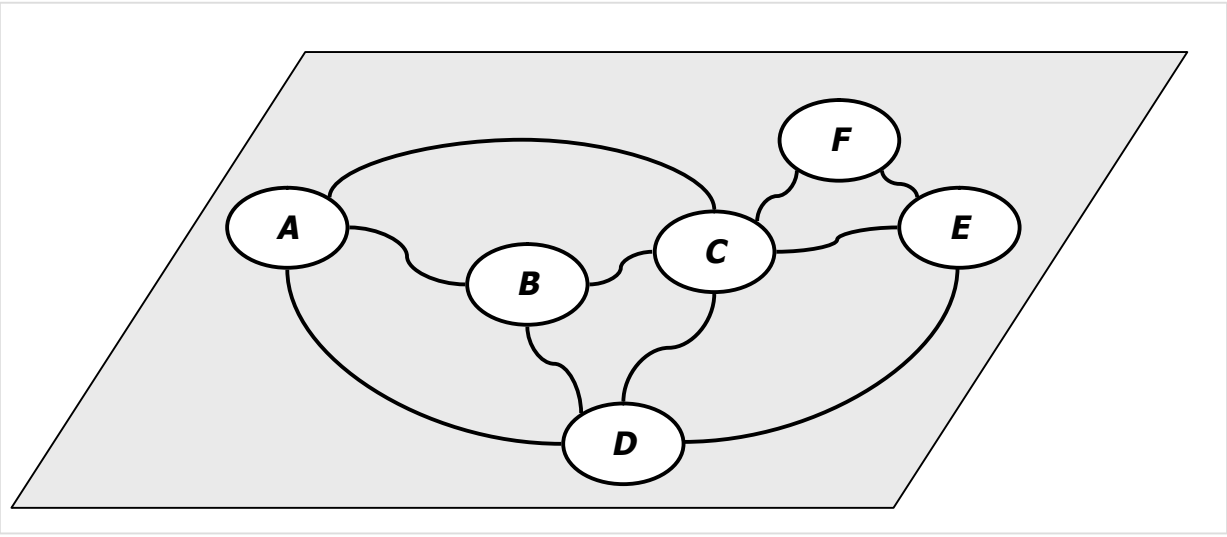


Рис. 31-4 — Граф, вершины у рёбра которого лежат на плоскости

Мы видим, что к вершине C плотно прижато несколько соседей, и потому алгоритм Прима с большой вероятностью построит дерево с узлом в вершине C, — такое дерево не будет гамильтоновым путём (рис. 31-5).

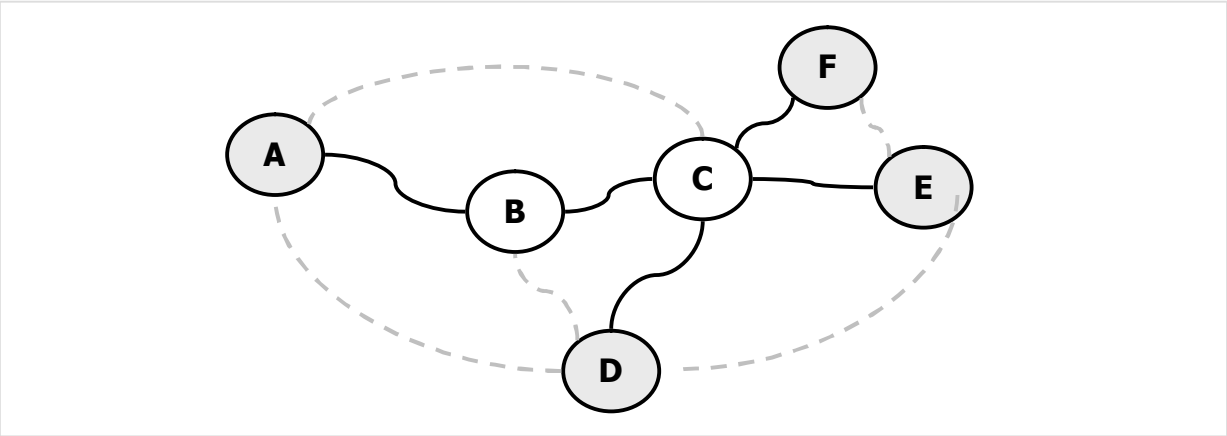


Рис. 31-5 — Минимальное покрывающее дерево с узлом C и четырьмя листьями

Теперь немного деформируем граф, приподняв вершину *C* над плоскостью так, чтобы длина инцидентных ей рёбер увеличилась на одну и ту же величину **D** (рис. 31-6).

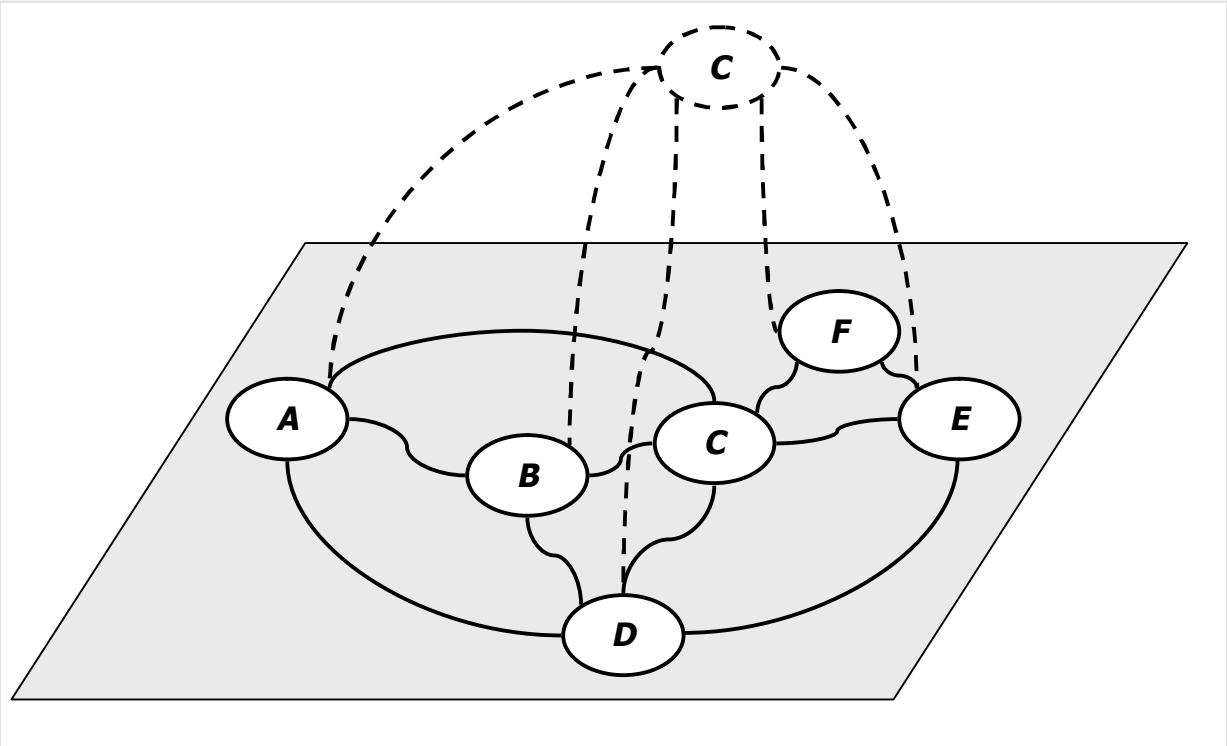


Рис. 31-6 — Граф с оштрафованной (приподнятой) вершиной C

«Скормив» этот деформированный граф алгоритму Прима, мы с некоторой вероятностью можем получить результат, показанный на рис. 31-7, — здесь покрывающее дерево выродилось в гамильтонов путь. Случилось так потому, что, удлинением инцидентных приподнятой вершине рёбер мы сделали их менее заманчивыми для Прима. И потому Прим оставил в дереве лишь кратчайшее из них, зато привлёк для постройки остова другие рёбра, и так бывший узел *C* стал листом, а бывшие листы *D*, *E* и *F* превратились в суставы.

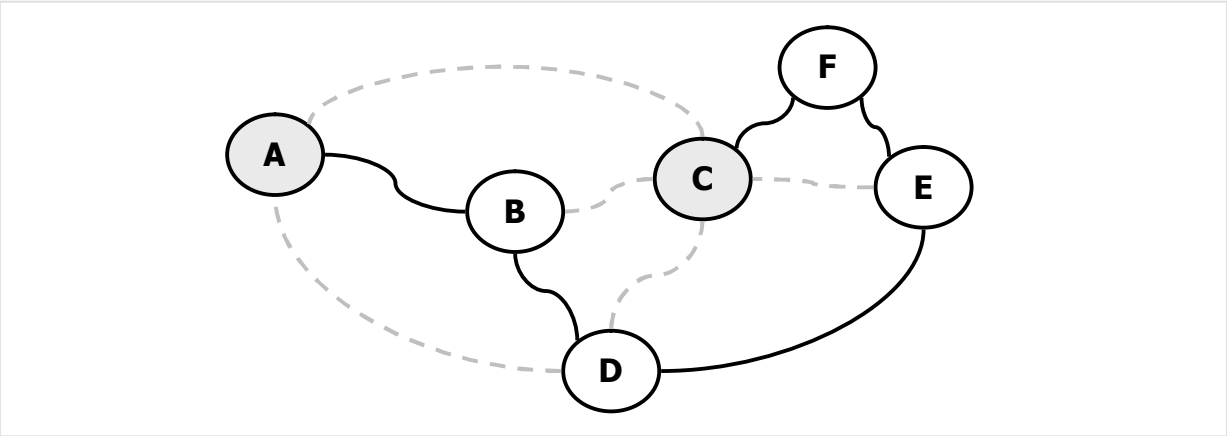


Рис. 31-7 — Покрывающее дерево, построенное после деформации исходного графа (A и C – листы, остальные вершины — суставы)

Итак, деформация исходного графа одновременным удлинением всех рёбер, инцидентных некоторому узлу, может изменить конфигурацию минимального покрывающего дерева, приблизив его форму к искомому гамильтонову пути. Разумеется, что стоимость деформированного дерева тоже изменится, — она увеличится. Но по исходным весам рёбер мы всегда сможем определить истинную стоимость этого подграфа. Настала пора разобраться в следующих вопросах:

- Как «удлинять» рёбра, не нарушая их исходной длины? Исходная длина нужна для определения стоимости гамильтонова пути.
- Почему поднятие вершин и «удлинение» рёбер в целом ведёт к решению задачи?
- На какую высоту «приподнимать» вершины (удлинять рёбра), чтобы найти решение по возможности быстро?

31.6.1. Способ «удлинения» рёбер

Вопрос с представлением «удлинённых» рёбер решается просто: «высоту» поднятия вершины будем отмечать так называемым *штрафом*, и хранить в поле вершины. Тогда «штрафная» длина некоторого ребра *A-B* определится сложением его исходной длины со штрафами двух его инцидентных вершин:

Штрафная длина *A-B* **= Исходная длина** *A-B* **+ штраф** *A* **+ штраф** *B*

В нашей программе штраф вершины будет храниться в поле **`TNode.mDist`**.

31.6.2. Сортировка остовных деревьев

Теперь покажем, как и почему штрафование вершин ведёт к точному или близкому к нему решению. В мысленном эксперименте представим себе все возможные остовы графа (пока все штрафы равны нулю). Расположим остовы в порядке их стоимости (длины) от самых «лёгких», до самых «тяжёлых», и будем наблюдать, как штрафование влияет на этот рейтинг-лист. Оштрафуем некую вершину *P* на величину **`D`**. Разумеется, что это увеличит веса всех деревьев, но в

разной степени, и потому порядок в «древесном» рейтинге изменится. Меньше других (на **1•D**) «потяжелеют» деревья, в которых оштрафованная вершина *P* является листом (1-вершиной). На **2•D** увеличатся веса деревьев, где она является суставом (2-вершиной), и так далее. И тогда в нашем воображаемом рейтинге остовы, где оштрафованная вершина является *узлом*, опустятся, а те, где она служит *листом* или *суставом* — соответственно поднимутся.

Теперь рассмотрим, каковы будут последствия от крупного штрафа начальной и конечной вершин *X* и *Y*. Напомню, что по условиям задачи они должны стать крайними вершинами, то есть, листьями. Огромный штраф отдалит эти вершины так далеко и друг от друга, и от прочих вершин, что модифицированный алгоритм Прима в кратчайшем остове сделает их *листами*, и соответствующая группа деревьев, где эти вершины являются листьями, *поднимется* в топ нашего рейтинга, — в этом состоит первый шаг будущего алгоритма.

Где-то в этой «всплывшей» группе деревьев окажется искомый нами *минимальный* гамильтонов путь, а также *другие* гамильтоновы пути и узловатые образования с лишними листьями. Дабы отделить гамильтоновы пути от других деревьев — приподнять их в рейтинге — продолжим аккуратно штрафовать оставшиеся узлы — вершины со степенями более двух. Пусть во вновь полученном кратчайшем дереве мы нашли такой узелок и оштрафовали его на величину **D**. Тогда все деревья, где степень этой вершины окажется больше двух, опустятся в рейтинге, а те, где она равна двум или меньше, — поднимутся. И у алгоритма Прима появится шанс построить минимальный остов, где бывший *узел* станет *суставом*, а количество *листьев* уменьшится. Отметим, что в результате этого штрафования веса всех гамильтоновых путей увеличатся на одну и ту же величину **2•D**, и потому кратчайший из них в своём семействе по-прежнему останется кратчайшим. В конце концов, после нескольких таких итераций на вершине рейтинга может оказаться остов, все вершины которого, за исключением двух крайних, будут суставами. Если это случится, то дерево окажется гамильтоновым путём, причём кратчайшим.

31.6.3. Выбор шага штрафования и сходимость

Итак, повторением штрафования узлов и построением минимального остова мы пытаемся поднять искомую гамильтонову цепь на вершину «рейтинга» остовных деревьев. И пусть алгоритм Прима весьма быстр, неоправданное его повторение существенно замедлит работу. Каков оптимальный шаг приращения штрафов? Слишком осторожный штраф не распрямит узелок, и нам придётся повторно штрафовать его и обращаться к алгоритму Прима. Слишком крупный штраф превратит оштрафованный узел в лист, а узел перекочует в другую вершину, и в итоге всё повторится. Для вычисления оптимального штрафа каждого узла рассмотрим следующую идею.

На рис. 31-8 дано дерево, полученное на некотором этапе алгоритмом Прима. Здесь для узлов *B* и *C* надо вычислить минимальные штрафы. Идея состоит в том,

чтобы поочерёдно «разрывать» рёбра, инцидентные узлу, а образовавшиеся при этом два поддерева графа соединять другими рёбрами, не инцидентными штрафуемому узлу, так, чтобы приращение стоимости вновь образованного дерева в итоге оказалось минимальным. Например, после разрыва ребра *B-C* поддеревья можно соединять рёбрами *A-C*, *A-F*, *D-C*, *D-E*.

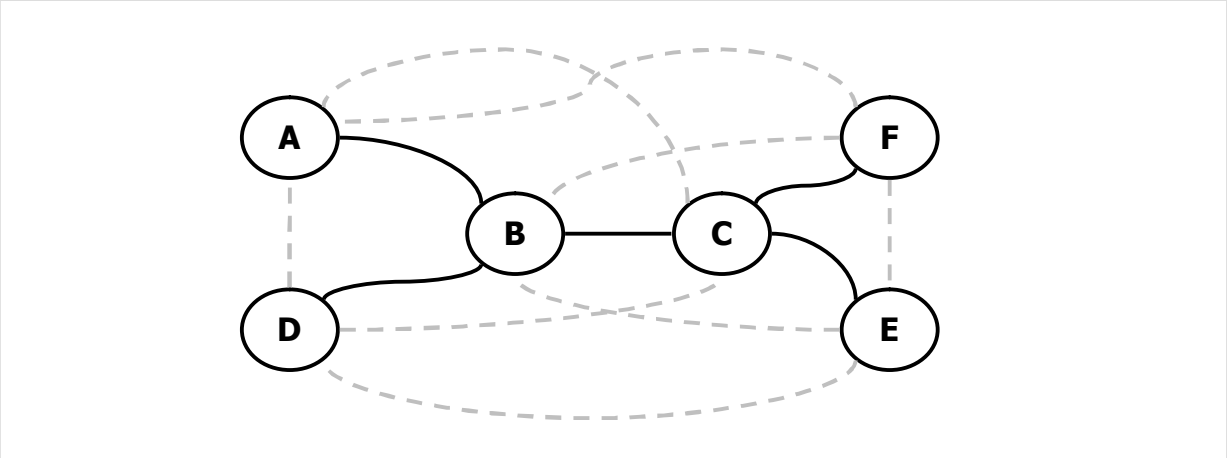


Рис. 31-8 — Остовное дерево с двумя узлами в вершинах B и C

Пусть кратчайшим из них оказалось ребро *A-C* (рис. 31-9), тогда минимальное приращение штрафа для разрыва ребра *B-C* составит:

```
Delta(B-C) = Cost(A-C) - Cost(B-C),
```

где **`Cost`** — функция, вычисляющая стоимость ребра с учётом текущих штрафов инцидентных ему вершин. Теперь, если оштрафовать вершину *B* на величину **`Delta`**, то штрафная цена ребра *B-C* сравняется со штрафной ценой ребра *A-C*, и оба они станут равно привлекательными для алгоритма Прима.

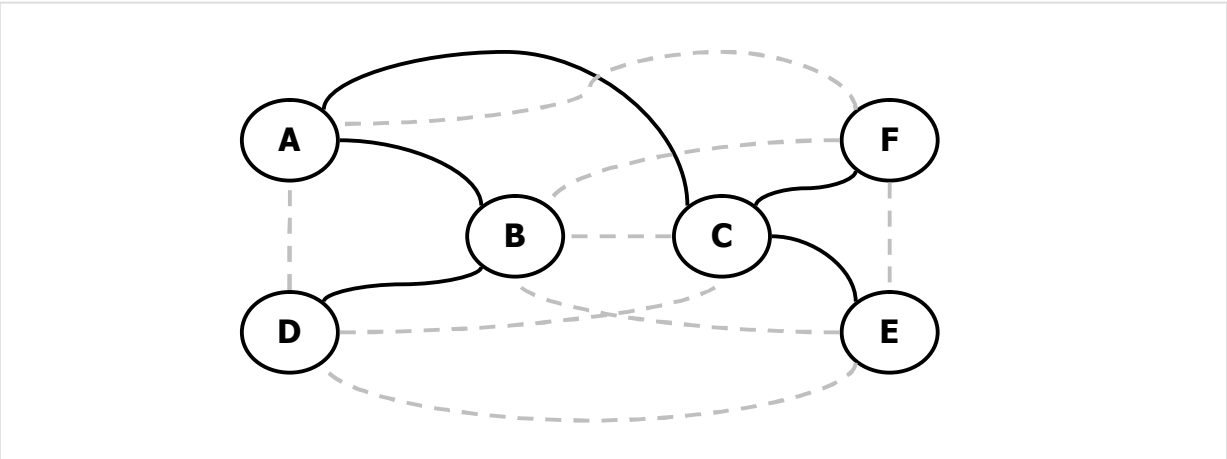


Рис. 31-9 — Разрыв ребра B-C и соединение поддеревьев ребром A-C

Так поочерёдно найдём приращения штрафов и для разрыва рёбер *A-B* и *B-D*, после чего выберем минимальное из трёх чисел и примем в качестве приращения штрафа для вершины *B*. Аналогично ищем приращение штрафа для вершины *C*.

Обработав, таким образом, все *узлы* текущего дерева, и назначив новые штрафы, мы вправе ожидать, что следующее применение алгоритма Прима породит остов с меньшим количеством узлов и листьев. Увы, ожидание это оправдывается не всегда ввиду сложного взаимного влияния штрафов разных вершин. Невязка порой не только не уменьшается, но даже увеличивается. И всё же после нескольких итераций тенденция к уменьшению невязки проявляется. Но только тенденция. Со временем обычно наблюдается эффект кочующих узлов: после очередной итерации узел переходит с одной вершину на другую, а затем обратно, что похоже на многократное повторение позиций в шахматной партии. Ничья, равносильная поражению? — не торопитесь капитулировать, выход есть. Решим ещё *два* вопроса: когда прекратить итерации, и что делать с оставшимися узлами? Но сначала «разрубим» узлы.

31.6.4. «Разрубание» узлов

Хотя разумное штрафование вершин и ведёт к постепенному повышению рейтинга искомого нами минимального гамильтонова пути, однако он может так и не «всплыть» на вершину рейтинга (что сразу дало бы точное решение). Вместо этого на верхушке оказываются деревья, по форме и весу более или менее близкие к гамильтонову пути. Почему бы здесь не «развязать» оставшиеся узелки что называется «в лоб»? Сделаем это способом, очень похожим на тот, что применён для вычисления штрафов. Будем поочерёдно разрывать рёбра, инцидентные узлу, а получившиеся поддеревья соединять лист к листу через кратчайшие рёбра.

На рис. 31-10 дан пример дерева с двумя узлами B и C, и двумя избыточными листьями D и E (невязка равна двум).

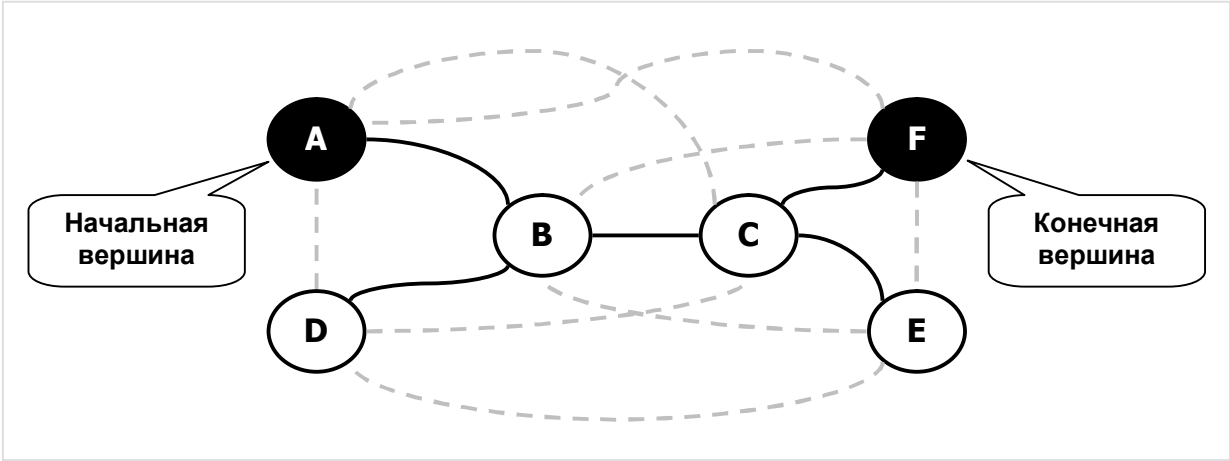


Рис. 31-10 — Исходное дерево с двумя узлами (невязка равна двум)

Для оптимального «разрубания» узла B поочерёдно разрываем инцидентные ему рёбра, а получающиеся при этом поддеревья соединяем «лист в лист». Так, после разрыва ребра A-B надо соединить поддерево A с остальными вершинами, и здесь кратчайшим мостом может оказаться, например, ребро A-D (рис. 31-11).

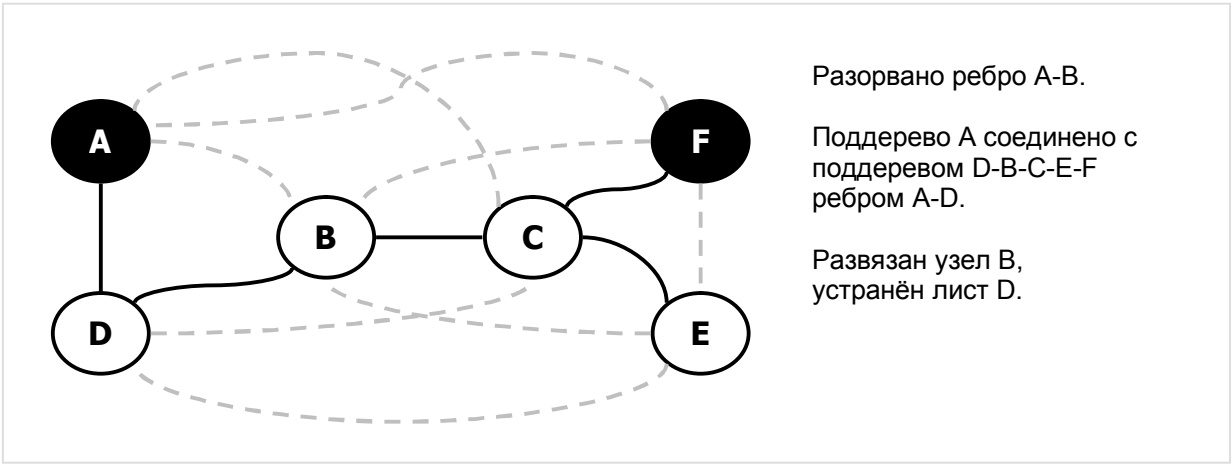


Рис. 31-11 — Разрыв ребра A-B и включение ребра A-D

В другой ситуации, после разрыва ребра *B-C*, кратчайшим соединением листа с листом может оказаться ребро *D-E* (см. рис. 31-12 — здесь устраняются сразу два узла и два листа). Перебрав, таким образом, все варианты разрывов и соединений, выберем тот, который даёт наименьшее приращение стоимости. Аналогично устраняется узел *C*.

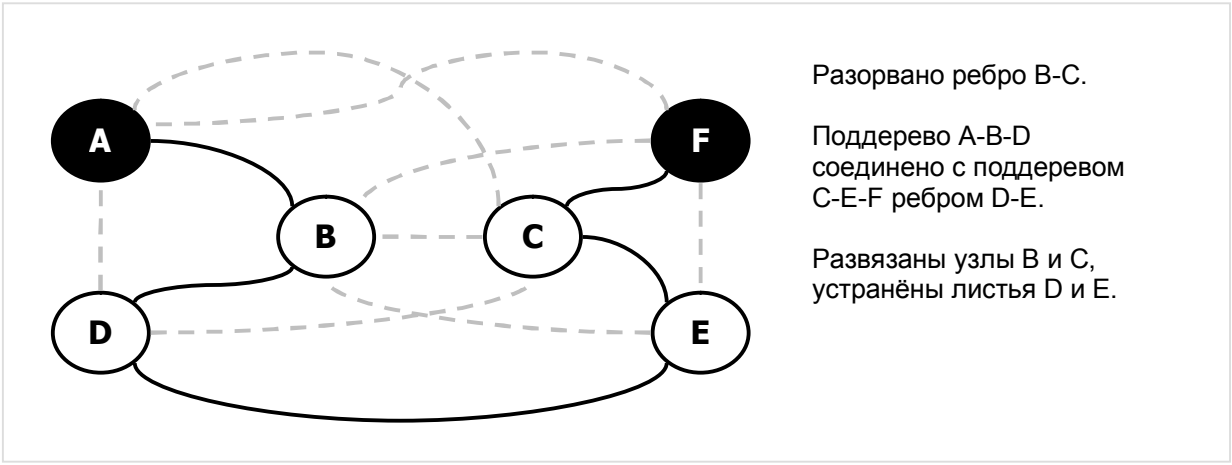


Рис. 31-12 — Разрыв ребра B-C и включение ребра D-E

Практика показала, что такое «силовое вмешательство» даёт цепь, весьма близкую к кратчайшему гамильтонову пути. После некоторого количества итераций (на разных остовах) эти манипуляции дают либо точный, либо близкий к нему результат.

31.6.5. Счётчики штрафования

Теперь обратимся ко второй проблеме: когда прекратить итерации и удовольствоваться полученным решением? Здесь возможны разные стратегии, и одна из них — простейшая — состоит в применении *счётчиков* штрафования вершин. Поскольку штрафование, в конечном счёте, ведёт к блужданию узлов, есть смысл ограничить число штрафов, налагаемых на каждую вершину. Учредим для каждой вершины счётчик, и при каждом штрафовании будем уменьшать его вплоть до нуля. Вершину с нулевым счётчиком далее не штрафуем. Когда после

некоторого числа итераций все *узлы* исчерпают свои счётчики, наложение штрафов прекратится (хотя запас штрафов в *листах* и *суставах* может сохраниться).

Проще всего исходные значения счётчиков задать *однократно*, но угадать это начальное значение трудно. А можно после каждой удачной итерации, приводящей либо к снижению невязки, либо к снижению стоимости очередного решения, *возобновлять* все счётчики, занося туда небольшие числа. Тем мы дадим алгоритму ещё несколько попыток улучшить решение. Рано или поздно, когда невязка и стоимость дерева перестанут снижаться, возобновление счётчиков прекратится, и в узлах дерева они будут исчерпаны.

В нашем алгоритме стратегия обновления счётчиков учтёт ещё и размер графа. Можно предположить, что в больших графах может потребоваться больше попыток штрафования. И потому изначально все счётчики устанавливаются равными **Count** = N, где N — число вершин графа. Затем после каждой удачной итерации счётчики всех верши возобновляются новым значением, вычисляемым из предыдущего по формуле:

Count = 2 + 3•Count/4

Возобновляемое значение счётчиков, таким образом, постепенно снижается, и затем остаётся таким до завершения алгоритма.

31.6.6. Алгоритм в целом

Мы обсудили основные идеи метода штрафования вершин, и теперь представим порядок действий в целом.

1. Установить начальное состояние всех вершин: цвет белый, штрафы нулевые, счётчики штрафования ненулевые, приращение счётчиков **Count:= N**.

2. Для исходной и конечной вершин назначить столь большие штрафы, чтобы гарантировано превратить их в листья.

3. Начало цикла.

    a. Методом Прима (модифицированным) построить покрывающей остов минимальной стоимости. Если дерево не получено, то решения нет.

    b. Подсчитать степени вершин в остове и невязку через число лишних листьев.

    c. Исходя из текущей конфигурации остова, подсчитать новые штрафы в узлах и количество штрафуемых узлов **Nodes**. Здесь **Nodes** будет равно нулю, когда невязка равна нулю, или когда исчерпаны счётчики штрафования в узлах.

d. Если в остове остались узлы, то «разрубить» их, получив очередной гамильтонов путь, близкий к искомому или совпадающий с ним. Если путь оказался лучше (дешевле) предыдущего, то запомнить его.

e. Если удалось улучшить решение или снизить невязку, то переустановить счётчики штрафования для всех вершин новым значением: **`Count = 2 + 3•Count/4`**.

f. Если количество оштрафованных вершин **`Nodes`** равно нулю (см. пункт **`c`**), то завершить цикл.

4. Если решение существует, то преобразовать множество рёбер дерева в гамильтонов путь (последовательность вершин).

В следующем пункте рассмотрим код, реализующий этот алгоритм. Он, наряду с обсуждёнными параметрами, будет принимать ещё один: так называемую *исключённую* вершину. Эта параметр, если он не равен **`NIL`**, указывает вершину, исключаемую из обработки алгоритмом. Мы вернёмся к пояснению этого параметра в следующей главе при поиске контура Гамильтона.

31.7. Ускоренный приближённый метод GenHamPathStrip

31.7.1. Вспомогательный объект

Перед тем, как представить метод, рассмотрим вспомогательный объект **`THamLink`**, необходимый для составления минимального остова с учётом штрафов вершин (листинг 31-3). Он будет элементом приоритетной очереди, упорядоченной в порядке не убывания «штрафной» длины рёбер.

Листинг 31-3 — Вспомогательная функция и объект для представления оштрафованных рёбер

```
// Вспомогательная функция для вычисления стоимости линка
// с учётом штрафов инцидентных вершин

function CalcValue(aLink: TLink): integer;
begin
  with aLink do Result:= mValue + mOwner.mDist + mDest.mDist;
end;


type  // THamLink -- вспомогательный линк для построения остова

  THamLink = class (TItem)
    mValue: integer; // стоимость с учётом штрафов
    mLink: TLink;    // ссылка на исходный линк
    constructor Create(aLink: TLink);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;

constructor THamLink.Create(aLink: TLink);
begin
```

```
  inherited Create;
  mLink:= aLink;
  // Условная стоимось формируется с учётом штрафов
  mValue:= CalcValue(aLink);
end;

function THamLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  with mLink do begin
    // Для неориентированного графа сравниваем исходную и конечную вершины
    // на предмет встречных линков
    if not mOwner.mOwner.mDirect and  // если не орграф
      (mOwner=(arg as THamLink).mLink.mDest) and
      (mDest=(arg as THamLink).mLink.mOwner)
      then begin
        // Здесь линки направлены встречно, отвергаем дубликат:
        Result:= cmpEq;    Exit;
      end;
  end;
  // Если линки не совпадают, то сортируем по неубыванию стоимости
  Result:= cmpLess;
  if mValue > (arg as THamLink).mValue then Result:= cmpGreate;
end;
```

31.7.2. Метод GenHamPathStrip

В следующем листинге дан метод (функция) приближённого ускоренного поиска кратчайшего гамильтонова пути между вершинами **aStart** и **aFin** с (возможным) исключением вершины **aStrip**. Функция возвращает последовательность вершин, составляющих путь, и стоимость пути.

Листинг 31-4 — Поиск гамильтонова пути методом штрафования вершин

```
function TGraph.GenHamPathStrip(aStart,            // начальная вершина
                                aFin,              // конечная вершина
                                aStrip: TNode;     // исключаемая вершина
                                var aCost: integer // длина (стоимость) цепи
                                ): TBuffer;

// Поля вершин TNode используются так:
// mDist    -- текущий штраф
// mPower   -- текущая степень вершины в остове
// mColor   -- текущая окраска
// mRoot    -- принадлежность поддереву
// mFlow    -- для временного хранения приращения штрафа
// mLimit   -- обратный счётчик штрафования (до нуля)

var   Que : TSet; // очередь рёбер по неубыванию стоимости с учётом штрафов
     Tree : TSet; // множество чёрных рёбер покрывающего дерева
   Marked : TBuffer;  // очередь вершин для маркировки поддеревьев

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка штрафных полей mDist (вызывается единожды)
```

```
  procedure ClearFines;
  var Node: TNode;      // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mDist:= 0;    // штраф
        mLimit:= 1;   // ненулевой предел на штрафование вершины
        mRoot:= nil;  // корневая метка
      end;
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возобновление счётчиков штрафования Node.mLimit
  // Вызывается после уменьшения невязки или стоимости

  procedure SetLimits(aLimit: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mLimit:= aLimit; // предел на штрафование вершины (счётчик)
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета вершин и рёбер, а также степеней вершин
  // (вызывается на каждой итерации перед построением дерева)
  // mColor = CWhite   -- цвет белый
  // mPower = 0        -- степень вершины в дереве

  procedure ResetNodesAndLinks;
  var Node : TNode;
      Link : TLink;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mColor:= CWhite; // цвет
      Node.mPower:= 0;      // степень вершины в покрывающем дереве
      // Сбросить цвета рёбер
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mColor:= CWhite;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
    // Исключаемую вершину пометить красным:
    if Assigned(aStrip) then aStrip.mColor:= CRed;
  end;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная процедура построения минимального покрывающего дерева
  // Формирует множество чёрных рёбер дерева и помещает в множество Tree

  function GenCover: boolean;

  var Node : TNode;     // текущая вершина
      Link : TLink;     // для пометки встречного линка
        HL : THamLink;  // ребро для дерева
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
    // Добавление в очередь Que линков к соседним белым вершинам.
    // Линки HL сортируются в Que по неубыванию штрафной длины

    procedure AddTreeLinks(aNode: TNode);
    var  Link : TLink;
           HL : THamLink;
    begin
      // Обработка исходящих связей
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) do begin
        // Вставить только линки, ведущие к белым вершинам
        if Link.mDest.mColor = CWhite then begin
          // Сконструировать вспомогательный линк:
          HL:= THamLink.Create(Link);
          // И вставить в очередь:
          if not Que.Insert(HL) then HL.Free;
        end;
        Link:= aNode.OutLinkNext;
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  var N : integer;  // количество присоединяемых вершин

  begin  { GenCover }
    Result:= true;
    // Предварительная очистка:
    Tree.ClrAndDestroy;   // дерево результата
    ResetNodesAndLinks;   // очистка вершин и рёбер: mColor = CWhite
    Node:= aStart;        // начинаем с исходной вершины
    Node.mColor:= CBlack; // красим чёрным
    // Присоединить все вершины графа кроме красной
    // (линков будет на единицу меньше)
    N:= mNodes.GetCount-1;
    if Assigned(aStrip) then Dec(N); // красную обойти стороной
    while Assigned(Node) and (Tree.GetCount < N) do  begin
      // Добавить в очередь Que линки к ближайшим белым вершинам:
      AddTreeLinks(Node);
      // Перебрать линки Que в поиске ближайшей белой вершины
      Node:= nil;                     // искомая вершина пока не найдена
      HL:= Que.GetFirst as THamLink;  // первый линк -- кратчайший
      while Assigned(HL) do begin
        Que.Delete(HL);               // удалить линк из очереди
        // Если линк ведёт к белой вершине, то ближайшая найдена
        with HL.mLink do if mDest.mColor = CWhite
            then Node:= mDest;    // приёмник связи
        if Assigned(Node) then begin
          // Здесь ближайшая белая вершина найдена:
          Node.mColor:= CBlack;     // присоединить её к множеству чёрных
          Tree.Insert(HL);          // вставить линк в результат (дерево)
          HL.mLink.mColor:= CBlack; // и отметить чёрным
          // Отметить чёрным и встречный линк:
          Link:= HL.mLink.GetReverse;
          if Assigned(Link) then Link.mColor:= CBlack;
          Break; // выход из while
        end else begin
          // Этот линк не ведёт к белой вершине, уничтожить его
          HL.Free;
          HL:= Que.GetNext as THamLink; // и взять следующий
        end;
      end; // while
    end; // while
    // Если остов не построен, то граф не связан, и решения не существует
```

```
    if Tree.GetCount < N then begin
      Tree.ClrAndDestroy;
      Result:= false;
    end; // if
    Que.ClrAndDestroy;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт степеней всех вершин, помещение их в поля TNode.mPower
  // и подсчёт невязки через количество листьев

  function CalcPowerNodes: integer;
  var HL : THamLink;   // текущий линк дерева Tree
      Node: TNode;
  begin
    // Предварительно очистить поля mPower
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // степень вершины в покрывающем дереве
      Node:= NodeNext;
    end;
    // Перебор линков в дереве Tree:
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      // наращиваем степени вершин дерева :
      Inc(HL.mLink.mOwner.mPower);     // в источнике дуги
      Inc(HL.mLink.mDest.mPower);      // в приёмнике дуги
      HL:= Tree.GetNext as THamLink;
    end;
    // Подсчитать невязку как сумму листьев минус 2
    Result:=-2;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mPower=1 then Inc(Result);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Маркировка поддерева из данной вершины (метим поле mRoot).
  // Дерево обходим в ширину по чёрным линкам.
  // Маркированные аершины красим серым.

  procedure Mark(aNode: TNode);
  var   Link : TLink;
        Node : TNode;
  begin
    Marked.Clear;          // очередь вершин для маркировки поддеревьев
    Marked.Put(aNode);     // занести в очередь корневую вершину
    aNode.mColor:= CGray; // и окрасить серым
    // Пока очередь не пуста:
    while Marked.GetCount > 0 do begin
      Node:= Marked.Get as TNode;    // выбрать вершину из очереди
      Node.mRoot:= aNode;            // и отметить её в поле mRoot
      Link:= Node.OutLinkFirst;
      // Поместить в очередь соседние вершины, связанные чёрными линками:
      while Assigned(Link) do begin
        // Если линк чёрный
        if (Link.mColor = CBlack) and
           // и цвета инцидентных вершин не совпадают
           (Node.mColor <> Link.mDest.mColor) then begin
          // то присоединить соседа:
          Marked.Put(Link.mDest);      // в очередь
          Link.mDest.mColor:= CGray;   // и окрасить серым
```

```
      end;
      Link:= Node.OutLinkNext;
    end;
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисление минимального штрафа в результате разрыва ребра aLink
// Вызывается из CalcNodeFine
// Возвращает штраф и признак успеха aStatus

function CalcLinkFine(aLink: TLink; var aStatus: boolean): integer;
var Link : TLink;
    Node : TNode;
    Fine : integer;
begin
  Result:= MaxInt;  aStatus:= false;
  // В текущий момент все вершины окрашены чёрным, а линки так:
  // принадлежащие дереву - чёрным, разорваный - серым, прочие - белым
  // Пометить вершины двух поддеревьев (поля mRoot)
  Mark(aLink.mOwner);  // поддерево mOwner
  Mark(aLink.mDest);   // поддерево mDest
  // Найти ребро минимального веса, соединяющее два поддерева
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Восстановить цвет после маркировки (кроме красной вершины):
    if Node <> aStrip then Node.mColor:= CBlack;
    // Если это вершина поддерева mOwner и она не корневая
    if (Node.mRoot = aLink.mOwner) and (Node <> aLink.mOwner) then begin
      // то обрабатать её, перебирая исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Если ребро ведёт в другое поддерево,
        // то запомнить его минимальный вес с учётом штрафа:
        // Link.mDest.mRoot -- метка вершины
        // aLink.mDest -- корень оторванной части
        if Link.mDest.mRoot = aLink.mDest then begin
          Fine:= CalcValue(Link);
          if Result > Fine then begin
            Result:= Fine;  // предварительный результат
            aStatus:= true; // признак наличия результата
          end;
        end;
        Link:= Node.OutLinkNext;
      end;
    end;
    Node:= NodeNext;
  end;
  // Окончательный результат (приращение штрафа)
  // вычислить как разность
  if aStatus then Result:= Result - CalcValue(aLink);
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Подсчёт минимального приращения штрафа для данной вершины
// вызывается из CalcFines
// Возвращает штраф и признак успеха aStatus

function CalcNodeFine(aNode: TNode; var aStatus: boolean): integer;
var Link, Reverse: TLink;  // прямой и обратный линки
    Fine: integer;         // вычисленный штраф
    OK: boolean;           // признак успеха
begin
  Result:= MaxInt;  aStatus:= false;
```

```
    // Поочерёдно "разрываем" линки вершины, входящие в состав дерева
    // (чёрные линки) и строим обходные пути,
    // соединяющие разорванные части дерева
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) and (Result > 0) do begin
      if Link.mColor = CBlack then begin
        // Этот линк принадлежит дереву
        // Временно "разорвать" его и встречный линк (красить серым)
        Link.mColor:= CGray;
        Reverse:= Link.GetReverse;
        if Assigned(Reverse) then Reverse.mColor:= CGray;
        // Вычислить штраф:
        aNode.OutPosPush;   // сохранить позицию перебора
        Fine:= CalcLinkFine(Link, OK); // вычислить штраф
        aNode.OutPosPop;    // восстановить позицию перебора
        // Вновь восстанавить цвета прямого и встречного линков
        Link.mColor:= CBlack;
        if Assigned(Reverse) then Reverse.mColor:= CBlack;
        // Запоминаем минимальное приращение штрафа
        if OK and (Result > Fine) then begin
          Result:= Fine;
          aStatus:= true;
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт и установка штрафов для всех вершин
  // Возвращает количество оштрафованных вершин

  function CalcFines: integer;
  var Node: TNode;
      Fine: integer;
      OK: boolean;
  begin
    Result:=0; // для подсчёта оштрафованных вершин
    // Сканировать вершины, степень которых превышает 2,
    // вычислять приращения штрафов и временно помещать в поля mFlow
    // Node.mPower -- степень вершины в дереве
    // Node.mFlow -- сюда временно помещаем приращение штрафа
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= 0;          // mFlow -- приращение штрафа
      if (Node.mPower > 2)     // если это узел
         and (Node.mLimit > 0)// и лимит штрафования не исчерпан
        then begin
        Dec(Node.mLimit); // уменьшить лимит
        Inc(Result);      // нарастить счётчик оштрафованных
        // Вычисляить минимальный штраф, уменьшающий её степень:
        PosPush; // сохранить позицию перебора вершин
        Fine:= CalcNodeFine(Node, OK);
        PosPop;  // восстановить позицию перебора вершин
        if OK then begin
          if Fine = 0 then Fine:=1;  // реальный штраф больше нуля
          Node.mFlow:= Fine;         // mFlow -- приращение штрафа
        end;
      end; // if
      Node:= NodeNext;
    end; // while
    // Нарастить штрафы вершин mDist:= mDist + Node.mFlow
    Node:= NodeFirst;
```

```
    while Assigned(Node) do begin
      Inc(Node.mDist, Node.mFlow);
      Node:= NodeNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вычисление минимального приращения цены
  // в результате разрыва ребра aLink
  // Вызывается из CalcDeltaNode
  // Возвращает кратчайший белый линк, соединяющий листья поддеревьев

  function CalcDeltaLink(aLink: TLink; var aRes: TLink): integer;
  var Link : TLink;
      Node : TNode;
  begin
    Result:= MaxInt;
    aRes:= nil;
    // В текущий момент все вершины окрашены чёрным,
    // линки дерева - чёрным, разорваный линк - серым, остальные - белым
    // Пометить вершины двух поддеревьев (поля mRoot)
    Mark(aLink.mOwner);  // поддерево mOwner
    Mark(aLink.mDest);   // поддерево mDest
    // Найти линк минимального веса, соединяющий два поддерева
    // через вершины со степенями 1 (листья)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Восстановить цвет после маркировки (кроме красной вершины)
      if Node <> aStrip then Node.mColor:= CBlack;
      // Если это лист в дереве mOwner
      if (Node.mRoot = aLink.mOwner) and (Node.mPower < 2)
        then begin
        // то обработать его
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // Запомнить кратчайший линк к листу другого поддерева
          if (Link.mDest.mRoot = aLink.mDest)  // в другом поддереве
            and (Link.mDest.mPower < 2)        // и степень меньше 2-х
            and (Result > CalcValue(Link))     // и штрафная длина меньше
          then begin
              Result:= CalcValue(Link);  // промежуточный результат
              aRes:= Link;               // белый линк
          end; // if
          Link:= Node.OutLinkNext;
        end; // while
      end; // if
      Node:= NodeNext;
    end; // while
    // Окончательный результат вычислить как разность
    if Assigned(aRes) then Result:= Result - CalcValue(aLink);
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск подходящей пары линков для снижения степени узла aNode
  // Вызывается из ForceCoverTree
  // Возвращает два линка: aBlack -- разрываемый, aWhite -- замещающий

  procedure FindBlackAndWhite(aNode: TNode; var aBlack, aWhite: TLink);
  var Link : TLink;   // прямой линк
      White: TLink;   // белый линк
      Delta: integer; // текущее приращение
      Minim: integer; // минимальное приращение
  begin
    aBlack:= nil;    aWhite:= nil;
```

```
    Minim:= MaxInt;
    // Поочерёдно "разрывать" линки вершины, входящие в состав дерева
    // (эти линки помечены чёрным)
    // и отыскивать обходные пути,
    // соединяющие разорванные части дерева через листья
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if Link.mColor = CBlack then begin
        // Этот линк принадлежит дереву, разорвать его (красим серым)
        Link.mColor:= CGray;
        // При разрыве уменьшаются степени инцидентных вершин:
        with Link do begin
          Dec(mOwner.mPower);
          Dec(mDest.mPower);
        end;
        aNode.OutPosPush; // сохр. позицию перебора
        // Вычислить приращение цены
        Delta:= CalcDeltaLink(Link, White);
        aNode.OutPosPop;  // восст. позицию перебора
        // Восстанавить цвет линка
        Link.mColor:= CBlack;
        // При воссоединении чёрного линка
        // восст. степени инцидентных вершин:
        with Link do begin
          Inc(mOwner.mPower);
          Inc(mDest.mPower);
        end;
        // Запомнить минимальное приращение
        if Assigned(White) and (Delta < Minim) then begin
          Minim:= Delta;   // приращение
          aBlack:= Link;   // чёрный линк (в дереве)
          aWhite:= White;  // белый линк (вне дерева)
        end;
      end;
      Link:= aNode.OutLinkNext;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление из дерева вспомогательных линков
  procedure Tree_Delete(aLink: TLink);
  var HL : THamLink; // линк в дереве Tree
      Rev: TLink;    // встречный линк
  begin
    // Уменьшить степени инцидентных вершин чёрного ребра aLink:
    Dec(aLink.mOwner.mPower);
    Dec(aLink.mDest.mPower);
    // Прямой и встречный линк метить белым
    aLink.mColor:= CWhite;
    Rev:= aLink.GetReverse;  // встречный линк
    if Assigned(Rev) then Rev.mColor:= CWhite;
    // Найти в дереве соответствующий вспомогательный линк
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      if (HL.mLink = aLink) or (HL.mLink = Rev) then Break;
      HL:= Tree.GetNext as THamLink;
    end;
    // Удалить его из дерева и ликвидировать
    if Assigned(HL) then begin
      Tree.Delete(HL);
      HL.Free;
    end;
  end;
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вставка в дерево вспомогательных линков

procedure Tree_Insert(aLink: TLink);
var HL : THamLink;
    Rev: TLink;
begin
  // Увеличить степени инцидентных вершин:
  Inc(aLink.mOwner.mPower);
  Inc(aLink.mDest.mPower);
  // Прямой и встречный линк метить чёрным
  aLink.mColor:= CBlack;
  Rev:= aLink.GetReverse;
  if Assigned(Rev) then Rev.mColor:= CBlack;
  // Сконструировать вспомогательный линк:
  HL:= THamLink.Create(aLink);
  // И вставить его в дерево:
  if not Tree.Insert(HL) then HL.Free;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Принудительное уменьшение степеней узлов остова

procedure ForceCoverTree;
var Node: TNode;
    Black, White : TLink;
begin
  // Обработать узлы (вершины со степенями более 2)
  // Node.mPower -- степень вершины в дереве
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mPower > 2 then begin
      // Вычислить минимальный штраф, уменьшающий её степень
      PosPush; // сохранить позицию перебора вершин
      FindBlackAndWhite(Node, Black, White);
      PosPop;  // восстановить позицию перебора вершин
      // Если пара рёбер обнаружена
      if Assigned(Black) and Assigned(White) then begin
        // Видоизменить дерево
        Tree_Delete(Black);   // удалить из дерева разрываемый линк
        Tree_Insert(White);   // вставить в дерево соединяющий линк
      end;
    end;
    Node:= NodeNext;
  end; // while
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Форсированное преобразование остова в цепочку
// с вычислением стоимости цепочки

function Force(var aCost: integer): TBuffer;

  // - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная функция для переноса цепи
  // из буфера aBuf в результат Result
  // с одновременным подсчётом стоимости цепи

  function MoveResult(aRes: TBuffer): integer;
  var HL : THamLink; // текущий линк дерева из Tree
  begin
    Result:= 0;
    aRes.Clear;
    // Перебор линков в покрывающем дереве:
```

```
      HL:= Tree.GetFirst as THamLink;
      while Assigned(HL) do begin
        aRes.Put(HL.mLink);
        Inc(Result, HL.mLink.mValue); // накопление стоимости
        HL:= Tree.GetNext as THamLink;
      end;
    end;

  var  Delta : integer;  // невязка, вычисляемая по числу листьев

  begin { Force }
    // Повторять, пока невязка больше нуля
    repeat
      // Форсировано снизить степени "узлов"
      ForceCoverTree;
      // Подсчитать степени вершин и невязку через количество листьев
      Delta:= CalcPowerNodes;
    until Delta=0;
    // Перенести цепь из буфера Buf в результат Result
    // и подсчитать стоимость цепи aCost
    Result:= TBuffer.Create;
    aCost:= MoveResult(Result);  //
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Преобразование буфера результата:
  // линки заменяются цепочкой вершин от Start до Fin

  function ConvertLinksToNodes(aLinks: TBuffer): TBuffer;

    // Поиск в буфере aLinks следующей за aNode вершины

    function FindNext(aNode: TNode): TNode;
    var Link: TLink;
        i: integer;
    begin
      Result:= nil;
      // Круговое вращение буфера в поиске нужного линка
      for i:= 1 to aLinks.GetCount do begin
        Link:= aLinks.Get as TLink;
        // Если линк найден, то цикл прекращается
        with Link do begin
          if aNode = mOwner then begin
            Result:= mDest;  Break;
          end;
          if aNode = mDest then begin
            Result:= mOwner; Break;
          end;
        end;
        // Если не тот линк, то вернуть его в буфер
        aLinks.Put(Link);
      end;
    end;

  var  Node: TNode;

  begin { ConvertLinksToNodes }

    Result:= TBuffer.Create;
    // Цепочка начинается с вершины aStart
    Node:= aStart;
    repeat
      Result.Put(Node);       // поместить в буфер
```

```
      Node:= FindNext(Node);  // и найти следующую
    until (Node = aFin) or not Assigned(Node);
    // Цепочка завершается вершиной aFin
    if Assigned(Node) then Result.Put(Node);
    aLinks.ClrAndDestroy;
    aLinks.Free;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    OK : boolean;        // признак связности графа
    Changed: boolean;    // признак снижения стоимости или невязки
    Delta : integer;     // невязка, вычисляемая по степеням вершин
    MinDelta : integer; // текущая минимальная невязка
    Nodes: integer;      // количество оштрафованных вершин
    Cost: integer;       // текущая стоимость
    BestCost: integer;   // стоимость лучшего промежуточного решения
    Res: TBuffer;        // очередное решение (цепь)
    FineLimit: integer; // предел штрафования вершины (счётчик)

begin  { TGraph.GenHamPath }

  aCost:= -1; // цена на случай отсутствия решения

  // Создать буфер для линков гамильтоновой цепи:
  Result:= TBuffer.Create;
  // Если это орграф, то выход:
  if mDirect then Exit;

  // Назначить начальное значение для счётчиков штрафования вершин
  FineLimit:= mNodes.GetCount;

  // Создать:
  Que:= CreateSet;           // множество для очереди линков
  Tree:= CreateSet;          // буфер для линков дерева
  Marked:= TBuffer.Create;  // очередь вершин для маркировки

  BestCost:= MaxInt;  // начальная лучшая стоимость

  // Очистить штрафные поля mFines:
  ClearFines;

  // Назначить максимальные штрафы (mDist) для крайних вершин:
  aStart.mDist:= MaxInt div 4;
  aFin.mDist:= aStart.mDist;

  MinDelta:= MaxInt;       // минимальная невязка

  // Повторять, пока можно оштрафовать вершины
  repeat
    // Построить кратчайшее покрывающее дерево с учётом штрафов:
    OK:= GenCover;
    // Выход, если граф не связан:
    if not OK then Break;
    // Подсчитать невязку Delta как сумму листьев - 2
    Delta:= CalcPowerNodes;
    // Подсчитать новые штрафов
    // на основе текущих степеней вершин (полей mPower)
    // вернуть количество оштрафованных вершин Nodes
    Nodes:= CalcFines;
    Changed:= false;   // признак снижения стоимости или невязки
    // Искать форсированное решение
    Res:= Force(Cost);
```

```
    // Если оно лучше текущего, то запомнить, а иначе удалить
    if Cost < BestCost then begin
      Changed:= true;   // признак снижения стоимости или невязки
      BestCost:= Cost;  // запомнить лучшую стоимость
      aCost:= Cost;     // и текущую стоимость
      Result.Free;      // прежний результат удалить
      Result:= Res;     // а новый сохранить
    end else Res.Free;  // а иначе удалить новый
    // При уменьшении невязки
    if Delta < MinDelta then begin
      Changed:= true;       // признак снижения стоимости или невязки
      MinDelta:= Delta;     // запомнить минимальную невязку
    end;
    // Если уменьшилась стоимость или невязка
    if Changed then begin
      SetLimits(FineLimit); // возобновить количество попыток штрафования
      FineLimit:= 2 + 3*FineLimit div 4; // пересчитать лимит штрафования
    end;
  until Nodes = 0;  // прекратить, если штрафование не выполнялось
  // По окончании преобразовать буфер результата:
  // линки --> в вершины от Start ... до Fin
  if OK then Result:= ConvertLinksToNodes(Result);
  // Очистить память:
  Marked.Free;         // очередь вершин для маркировки
  Tree.ClrAndDestroy; // остовное дерево
  Tree.Free;
  Que.Free;            // очередь линков
end;
```

Далее обсуждаются некоторые детали листинга.

31.7.3. Функция GenCover

Эта функция строит минимальный остов графа с учётом текущих штрафов, наложенных на вершины. Функция возвращает **TRUE**, если остов построен. В случае, когда параметром **aStrip** задана исключаемая вершина, функция заранее окрашивает её в запрещающий красный цвет, и тогда эта вершина вместе с инцидентными ей рёбрами, игнорируются.

31.7.4. Процедура Mark

Вызывается всякий раз при «разрыве» очередного ребра остова для маркировки вершин двух получаемых при этом поддеревьев. Одно из них метится вершиной-источником линка, второе — вершиной приёмником. Параметр процедуры содержит одну из этих вершин.

31.7.5. Функции CalcFines, CalcNodeFine и CalcLinkFine

Подсчитывают оптимальные штрафы соответственно всех вершин, отдельной вершины, отдельного линка.

31.7.6. Функция CalcDeltaLink

Возвращает кратчайший белый линк, соединяющий листья двух поддеревьев в процедуре «разрубания» узлов.

31.7.7. Функция Force и процедура ForceCoverTree

На каждой итерации принудительно «разрубают» узлы дерева.

31.7.8. Функция ConvertLinksToNodes

Вызывается на завершающем этапе для преобразования буфера результата **`Tree`**, где в ходе решения задачи формируется множество рёбер гамильтонова пути. Это множество рёбер надо затем превратить в последовательность вершин от **`aStart`** до **`aFin`**.

31.8. Метод GenHamPath

Этот метод использует основной метод, и применяется в тех случаях, когда исключаемая вершина **`aStrip`** не задана (листинг 31-5).

Листинг 31-5 — Функция поиска гамильтонова пути без исключённой вершины

```
function TGraph.GenHamPath(aStart,              // начальная вершины
                           aFin: TNode;         // конечная вершины
                           var aCost: integer   // длина цепи
                           ): TBuffer;
begin
  Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;
```

31.9. Программы для испытаний

Ниже представлены программы для сравнительных испытаний обсуждённых ранее алгоритмов построения гамильтоновых путей, а именно: 1) случайного перебора, 2) жадного перебора, 3) полного перебора и 4) метода штрафования вершин. Алгоритмы сравниваются по точности результата и скорости. Результаты зависят и от характеристик обрабатываемых графов. Так, например, для графов с одинаковыми рёбрами все алгоритмы сформируют равноценные пути, — этот случай не интересен. Гораздо полезней испытать методы на полных графах, длины рёбер которых равномерно распределены на интервале от **`1`** до **`99`**.

Первая программа (листинг 31-6) сравнивает эффективность четырёх алгоритмов на отдельных неориентированных графах. Здесь графы формируются случайно, но могут быть взяты также из некоторого файла.

Листинг 31-6 — Программа сравнения алгоритмов на отдельных графах

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

var Gr : TGraph;
    Res: TBuffer;
    Cost: integer;
    Na, Nb, Nc : TNode;
    Start : TDateTime;
    Nodes, Cost1, Cost2, Time : integer;

// Преобразование буфера вершин в строку

function HamToStr(aBuf: TBuffer): string;
var N: TNode;
    i: integer;
begin
  Result:='';
  for i:= 1 to aBuf.GetCount do begin
    N:= aBuf.Get as TNode;
    aBuf.Put(N);
    Result:= Result+ N.GetName;
    if i<aBuf.GetCount then Result:=Result +'-';
  end;
end;

begin
  repeat
    Write(#10#10'Nodes= '); Readln(Nodes);
    if Nodes < 4 then Break;
    {
    Сформировать полный граф со случайным распределением длин рёбер
    в диапазоне от 1 до 99 (равномерное распределение)
    }
    Gr:= TGraphChars.GenFull(false, 0, 99, Nodes);
    if Nodes < 16 then Gr.Expo;
    {
    В качестве исходной и конечной вершин взять первую (A) и вторую (B)
    В качестве исключаемой вершины может быть взята вершина C
    }
    Na:= (Gr as TGraphChars).GetNode('A');
    Nb:= (Gr as TGraphChars).GetNode('B');
    //Nc:= (Gr as TGraphChars).GetNode('C');
    Nc:= nil;

    Writeln('========== Ham Path ==========='#10);
    Start:= Now;
    //Res:= Gr.GenHamPath(Na, Nb, Cost);
    Res:= Gr.GenHamPathStrip(Na, Nb, Nc, Cost1);
```

```
    Time:= MilliSecondsBetween(Start, Now);
    Writeln(HamToStr(Res));
    Writeln('Cost Ham  = ', Cost1:4, Time:7, #10);
    Res.Free;

    if Nodes < 18 then begin
      Writeln('========== Ham Path Full ==========='#10);
      Start:= Now;
      Res:= Gr.GenHamPath_Full(Na, Nb, Cost2);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Writeln('Cost Full = ', Cost2:4, Time:7, #10);
      Res.Free;
      Writeln('Ratio= ', 100*Cost1/Cost2 : 7:1);
    end;

    Writeln(#10'========== Ham Path Greed ==========='#10);
    Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, true); // жадно
    Writeln('Cost Greed = ', Cost);
    Res.Free;

    Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, false); // случайно
    Writeln('Cost Randm = ', Cost);
    Res.Free;
    Gr.Free;
  until false;
end.
```

Реалистичная картина достигается обработкой большого количества графов с накоплением статистики, что и выполняет следующая тестовая программа (листинг 31-7). Она определяет среднее время работы каждого метода, а также качество результата как отношение стоимости пути к стоимости идеального пути, полученного полным перебором. Для метода штрафования вершин формируется ещё и массив отклонений результата от идеала, — он пригоден для построения гистограммы.

Листинг 31-7 — Программа сравнительных испытаний на большом числе графов (формируется статистика)

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

const CPc = 2;       // шаг диаграммы в %
      CArray = 20;   // 2*20 = 40 %

var Gr  : TGraph;
    Res : TBuffer;
    Cost: integer;
    i,k : integer;
```

```
    Start : TDateTime;
    Time  : integer;

    Nodes : integer;  // количество вершин
    Retry : integer;  // количество повторений

    // Стоимости (%):
    CF : integer;   // полный перебор (100%)
    CQ : extended;  // быстрый
    CG : extended;  // жадный
    CR : extended;  // случайный

    // Времена:
    TF : integer; // полный перебор
    TQ : integer; // быстрый
    TG : integer; // жадный
    TR : integer; // случайный

    Arr : array [1..CArray] of integer; // массив для гистограммы
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вывод статистики
procedure ResultExpo(const aFile: String);
var n: integer;
begin
  if aFile <> '' then begin
    Assign(Output, aFile);
    if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
  end;
  Writeln('------------------------------------------------------------');
  Writeln('  Nodes=', Nodes:4, '  Retry=', Retry:4);
  Writeln('           %       ms');
  Writeln('Main =   100.0',           TF/Retry:7:0);
  Writeln('Quick= ',100*CQ/Retry:7:1, TQ/Retry:7:0);
  Writeln('Greed= ',100*CG/Retry:7:1, TG/Retry:7:0);
  Writeln('Rand = ',100*CR/Retry:7:1, TR/Retry:7:0);
  for n:= 1 to CArray do Writeln(n*CPc:3,#9, Arr[n]);;
  Writeln('------------------------------------------------------------');
  if aFile <> '' then begin
    Close(Output);  Assign(Output, '');    Rewrite(Output);
  end;
end;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var    Na, Nb : TNode; // начальная и конечная вершины

begin
  repeat
    Write('Nodes= '); Readln(Nodes);
    if Nodes<5 then Break;
    repeat
      Write('Retry= '); Readln(Retry);
      if Retry=0 then Break;
      // Очистка времён и стоимостей
      CQ:=0; CG:=0; CR:=0;
      TF:=0; TQ:=0; TG:=0; TR:=0;
      // Массив гистограммы
      for k:= 1 to CArray do Arr[k]:=0;

      for i:= 1 to Retry do begin
        Write('.');
        Gr:= TGraphChars.GenFull(false, 0, 99, Nodes);
        Na:= (Gr as TGraphChars).GetNode('A');
        Nb:= (Gr as TGraphChars).GetNode('B');
```

```
        //-----------------------------------------
        // Полный перебор
        Start:= Now;
        Res:= Gr.GenHamPath_Full(Na, Nb, Cost);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CF:= Cost; Inc(TF, Time);
        //-----------------------------------------
        // Ускоренный приближённый (штрафование вершин)
        Start:= Now;
        Res:= Gr.GenHamPath(Na, Nb, Cost);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CQ:= CQ + Cost/CF; Inc(TQ, Time);
        // k - индекс в массиве диаграммы
        k:= 1 + Round(100*(Cost/CF-1)/CPc);
        if k in [1..CArray] then Inc(Arr[k]);

        //-----------------------------------------
        // Жадный
        Start:= Now;
        Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, true);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CG:= CG + Cost/CF; Inc(TG, Time);
        //-----------------------------------------
        // Случайный
        Start:= Now;
        Res:= Gr.GenHamPath_Greed(Na, Nb, Cost, false);
        Time:= MilliSecondsBetween(Start, Now);
        Res.Free;
        CR:= CR + Cost/CF; Inc(TR, Time);
        //-----------------------------------------
        Gr.Free;
      end;
      Writeln;
      ResultExpo('');
      ResultExpo('Test_Time.txt');
    until false;
  until false;
end.
```

31.10. Результаты испытаний

В ходе испытаний обработаны по сотне случайных графов с количеством вершин от 10 до 17, результаты даны на следующих рисунках и в таблицах. В табл. 31-1 сопоставляется точность разных методов в сравнении с идеалом, получаемым полным перебором всех путей. Здесь быстрым алгоритмом назван метод штрафования вершин, он формирует очень близкие к идеалу пути (или совпадающие с ним), средняя цена которых лишь на **0,5%** превышает идеальную. В этом отношении жадный алгоритм значительно хуже метода штрафования, но много лучше случайного, что вполне естественно (см. рис. 31-13). Отметим, также, что качество этих алгоритмов — жадного и случайного — ухудшается с ростом размера графа.

Табл. 31-1 — Отношение стоимостей путей к идеалу, %

Количество вершин	Алгоритм		
	Быстрый	Жадный	Случайный
10	**100,5**	**134**	**256**
11	**101,0**	**133**	**280**
12	**100,5**	**140**	**305**
13	**100,7**	**140**	**328**
14	**100,3**	**147**	**356**
15	**100,6**	**150**	**396**
16	**100,3**	**155**	**392**
17	**100,3**	**153**	**425**
СРЕДНЕЕ ЗНАЧЕНИЕ	***100,5***	***144,0***	***342,3***

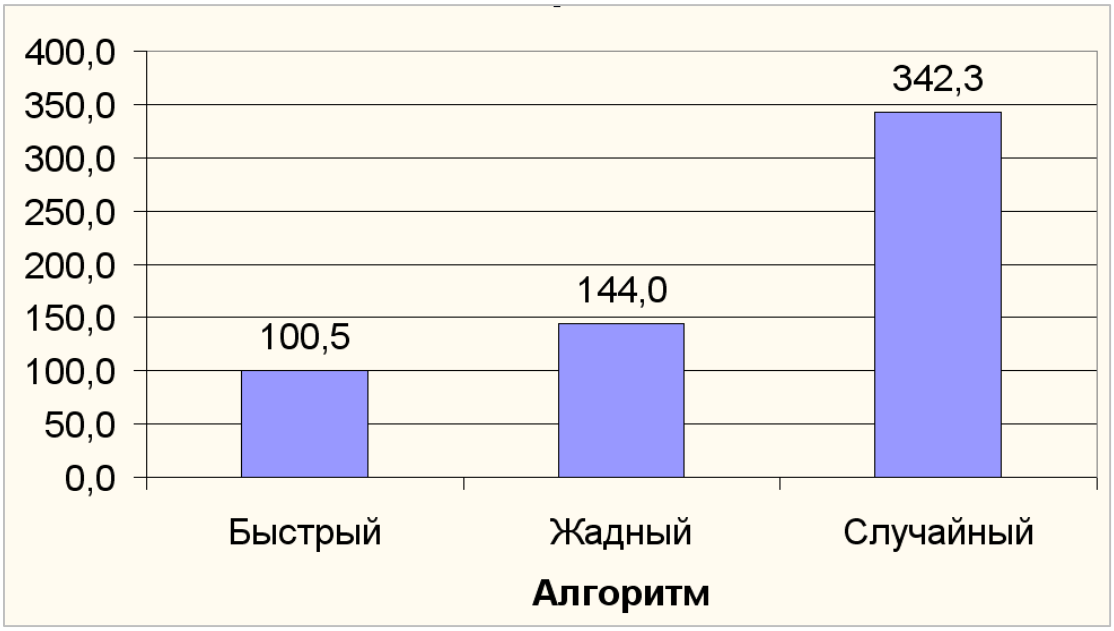


Рис. 31-13 — Отношение стоимостей путей, полученных разными алгоритмами, к стоимости идеального пути (%)

На рис. 31-14 дана гистограмма распределения стоимостей результатов быстрого алгоритма в сравнении с идеалом, — таблица получена по результатам 800 испытаний. Здесь видно, что в **87%** случаев результат быстрого алгоритма уступает идеалу не более **2%**, а в **97%** случаев — не более **6%**.

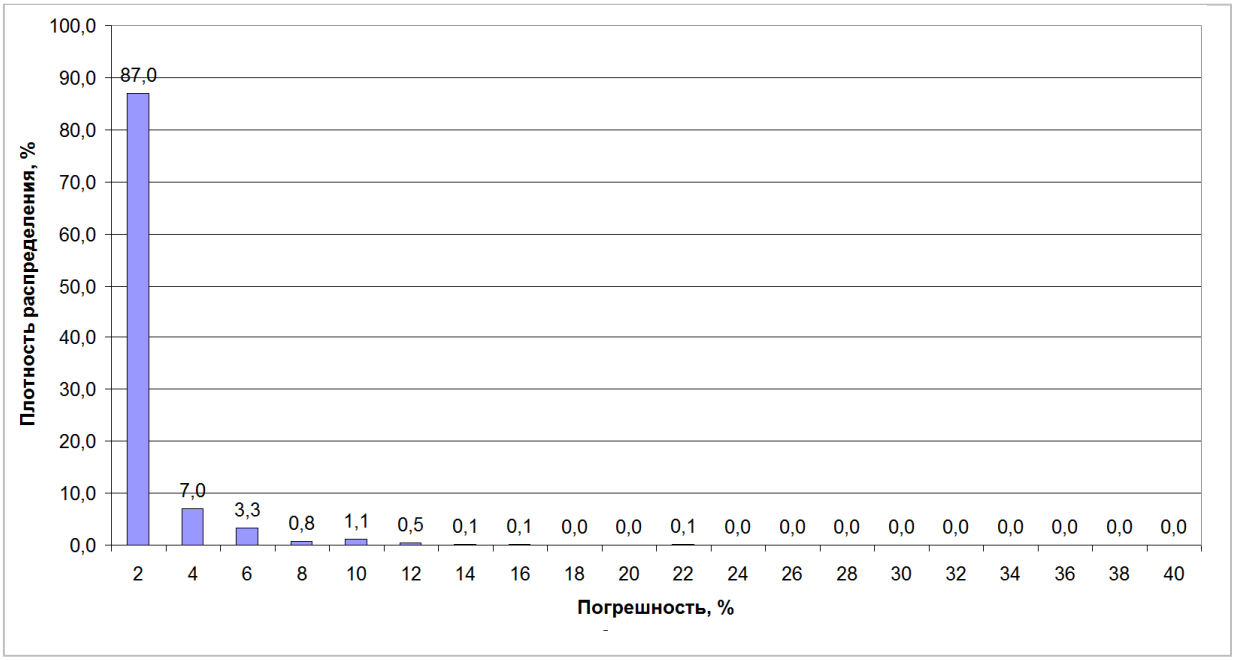


Рис. 31-14 — Плотность распределения результатов для быстрого алгоритма

Точность метода штрафования вершин признаем вполне приемлемой, а что со временем? В табл. 31-2 даны средние времена обработки графов точным алгоритмом полного перебора и быстрым методом штрафования вершин, а также соотношения этих времён. То же показано на рис. 31-15 и рис. 31-16.

Табл. 31-2 — Среднее время (ms) обработки одного графа полным перебором и быстрым методом, а также отношения времён

Количество вершин	Алгоритмы и отношение		
	Полный перебор T1, ms	Быстрый T2, ms	Отношение T1/T2
10	1,0	2	0,5
11	6,0	2	3,0
12	20,0	3	6,7
13	68,0	5	13,6
14	227,0	7	32,4
15	756,0	9	84,0
16	3002,0	13	230,9
17	6810,0	17	400,6

Эти данные подтверждают ценность быстрого алгоритма штрафования вершин, преимущество которого с ростом графа тоже растёт (рис. 31-16). При испытании отдельных графов наблюдается, также, большой разброс времени полного перебора разных графов с одинаковым количеством вершин (в разы). Скорость быстрого алгоритма для всех графов стабильна.

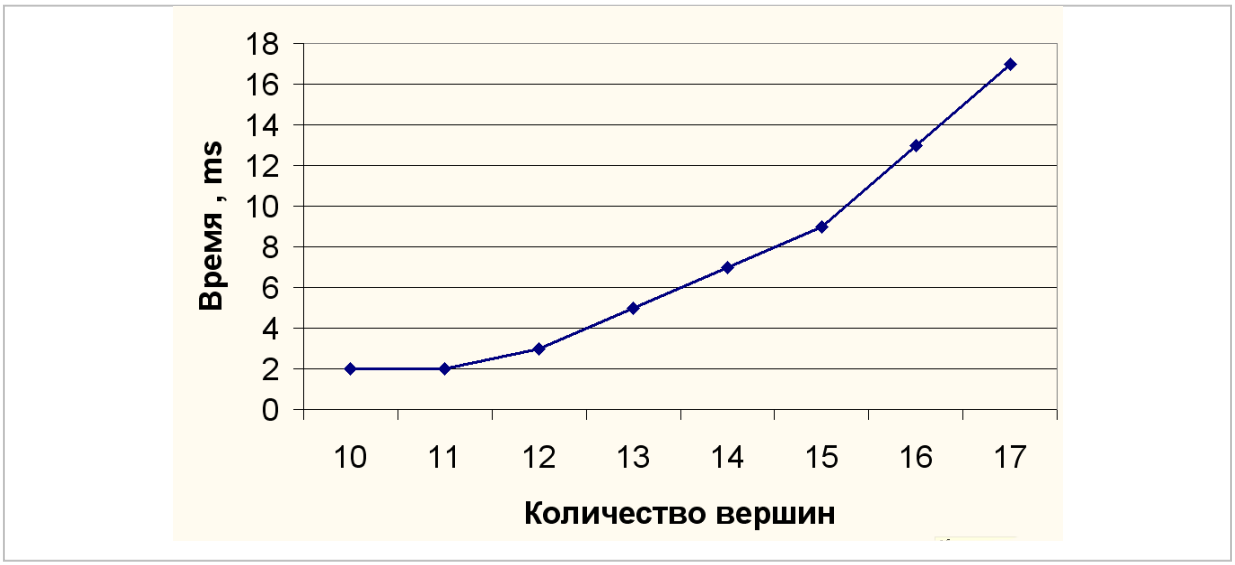


Рис. 31-15 — Среднее время обработки графа быстрым методом штрафования вершин

Рис. 31-16 — Отношение скорости быстрого алгоритма к скорости точного алгоритма (T1/T2)

31.11. Итоги

31.11.1. В задачах коммивояжёра и Гамильтона ищутся кратчайшие (или длиннейшие) пути посещения *всех* вершин графа.

31.11.2. В задачах коммивояжёра *разрешены* повторные посещения вершин, а в задачах Гамильтона — *нет* (каждая вершина посещается единожды).

31.11.3. Задачи посещения делят на *замкнутые* и *разомкнутые*.

31.11.4. В замкнутых задачах путь — контур — заканчивается в исходной вершине, тем самым исходной может быть любая вершина.

31.11.5. В разомкнутых задачах путь начинается и заканчивается в разных вершинах. Иногда интерес представляет пара вершин, путь между которыми минимален (или максимален).

31.11.6. Задачи коммивояжёра всегда имеют решения на связных графах и сильно связных орграфах, для задач Гамильтона этих условий недостаточно.

31.11.7. На *полных* графах всегда решаются и задачи коммивояжёра, и задачи Гамильтона.

31.11.8. Задачи коммивояжёра решают через задачи Гамильтона созданием вспомогательного полного графа, с последующим возвратом к исходному графу.

31.11.9. *Точное* решение задачи Гамильтона может быть получено перебором всех возможных путей, однако ввиду экспоненциально растущей трудоёмкости этот способ применим лишь для небольших графов.

31.11.10. *Быстрое*, но грубое решение даёт жадный алгоритм, выбирающий на каждом шаге кратчайшее ребро или дугу, ведущее в ещё не посещённую вершину.

31.11.11. Весьма *точное* и относительно *быстрое* решение для *неориентированных* графов даёт метод штрафования вершин. В его основе лежит многократное применение алгоритма Прима, строящего минимальный остов графа. Постепенно штрафуя вершины (взаимно отдаляя их друг от друга) алгоритм деформирует граф так, что кратчайший его остов становится одновременно и гамильтоновым путём, или максимально приближается к нему.

31.11.12. Быстрое и точное решение для ориентированных графов получается через решение замкнутой задачи о гамильтоновых путях (см. главу 33).

31.12. Задания

31.12.1. Задание 1

На основе метода штрафования вершин разработайте метод, отыскивающий в полном неориентированном графе кратчайший гамильтонов путь. Метод должен вернуть этот путь и его стоимость. Идея решения состоит в добавлении двух искусственных вершин, служащих исходной и конечной для искомого пути. Искусственные вершины связываются со всеми прочими вершинами через рёбра произвольной одинаковой длины. После отыскания кратчайшего пути между ними, искусственные вершины отбрасываются, а остаток будет кратчайшим из всех возможных гамильтоновых путей на данном графе.

31.12.2. Задание 2

Разработайте метод решения задач коммивояжёра на связном неориентированном графе. В основу положите формирование вспомогательного полного графа и отыскание на этом графе гамильтонова пути, после чего гамильтонов путь преобразуйте в путь коммивояжёра.

31.13. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 273
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 32
Замкнутая задача Гамильтона

Продолжим вникать в проблемы Гамильтона и обратимся к *замкнутому* варианту его задачи. Здесь начальной и конечной вершиной пути будет одна и та же (любая) вершина графа, стало быть, образуется замкнутый контур. От разомкнутого варианта задачи отличие, казалось бы, пустяшное — всего одно лишнее ребро или дуга, однако эта мелочь в корне меняет подход к решению.

32.1. Опорные алгоритмы

Как и для разомкнутого варианта, для оценки эффективности создаваемого алгоритма мы воспользуемся некими «опорными точками», то есть простыми алгоритмами, эффективными либо в отношении *быстродействия*, либо в отношении *точности*. Эти алгоритмы мало отличаются от своих аналогов для разомкнутого варианта, и потому здесь приведено лишь перечисление методов (тексты найдёте в приложении с исходными файлами):

`GenHamilton_Greed` — с параметром **`FALSE`** этот метод случайно выбирает очередную дугу, ведущую в неокрашенную вершину. Алгоритм быстрый, но не точный.

`GenHamilton_Greed` — с параметром **`TRUE`** выбирает кратчайшую дугу, ведущую в неокрашенную вершину (жадный алгоритм). По скорости почти не уступает случайному, но даёт существенно лучший результат, хотя и далёкий от идеального.

`GenHamilton_Full` — полный перебор всех возможных путей, даёт идеально точный результат, но крайне трудоёмок, и потому применим лишь для небольших графов.

Отметим, также, что случайный и жадный алгоритмы гарантируют решение лишь на *полных* графах.

32.2. Контур, поток и паросочетание

Воспринять основные идеи предстоящего решения будет проще после рассмотрения рис. 32-1, где показан полный орграф с уже обнаруженным гамильтоновым контуром минимальной длины (стоимости). Этот контур выделен жирным, а все прочие дуги графа слегка обозначены серым.

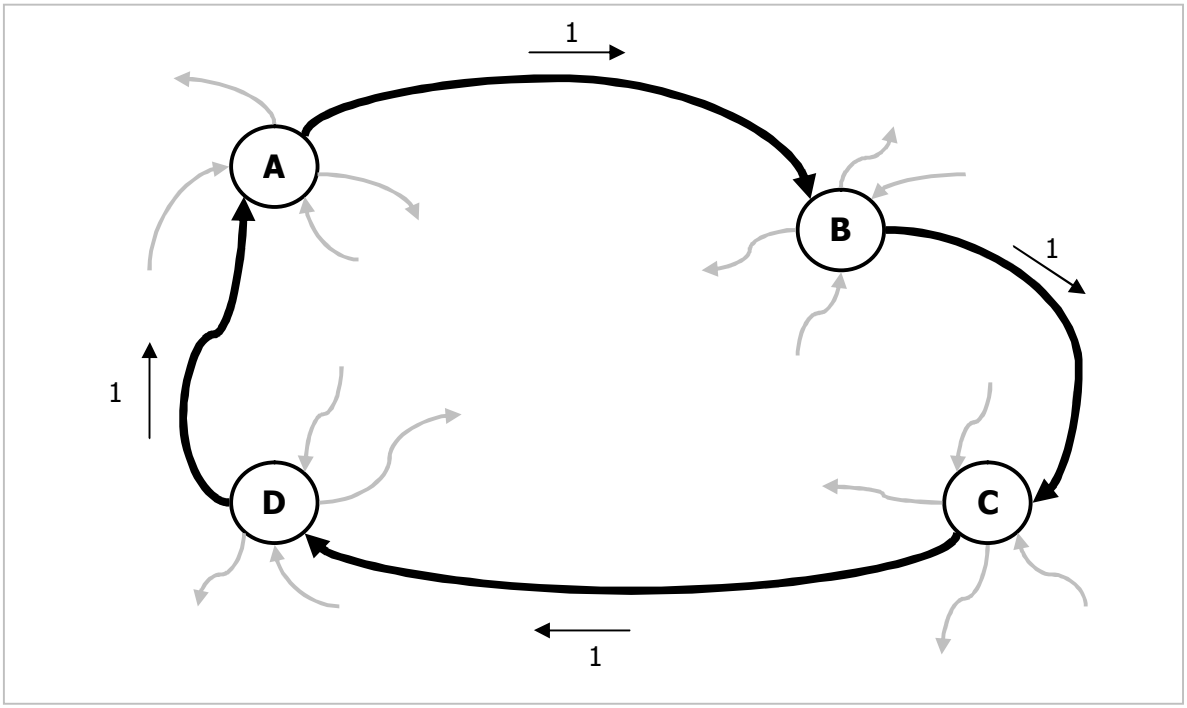


Рис. 32-1 — Единица потока, циркулирующая по замкнутому контуру

Пустим по этому замкнутому контуру некий единичный поток. Что можно сказать о его стоимости? Очевидно, что эта стоимость будет минимальной в сравнении со всеми прочими подобными потоками, проходящими через все вершины графа по иным дугам. Стало быть, задача поиска минимального гамильтонова контура родственна уже знакомой нам задаче поиска минимального потока в *разомкнутой* сети. Вопрос лишь в том, как найти этот минимальный поток в *замкнутом контуре*. Прибегнем к силе волшебства, и сведём эту задачу к поиску минимального потока в двудольном графе.

Этой волшебной силой разделим каждую вершину исходного графа на две половинки, одна из которых (**S**) будет содержать только исходящие дуги и служить источником потока, а другая (**T**) — только входящие дуги, и будет лишь принимать поток (рис. 32-2). Ясно, что вновь полученный граф будет двудольным, и количество вершин в нём окажется вдвое больше, чем в исходном. Представим его соответствующим рисунком (рис. 32-3). Из 27-й главы нам известны два способа поиска потока минимальной стоимости в таком графе. Один из них заключается в добавлении ещё двух вершин: общем истоке S и общем стоке T, которые соединяются со всеми прочими вершинами графа, как показано на рис. 32-3. Однако в той же главе мы выяснили, что эта задача равносильна поиску минимального паросочетания, и быстрее решается *венгерским* алгоритмом. Стало быть, найдя полное паросочетание минимальной стоимости на двудольном графе, мы тем самым найдём и контур минимальной стоимости в исходном графе.

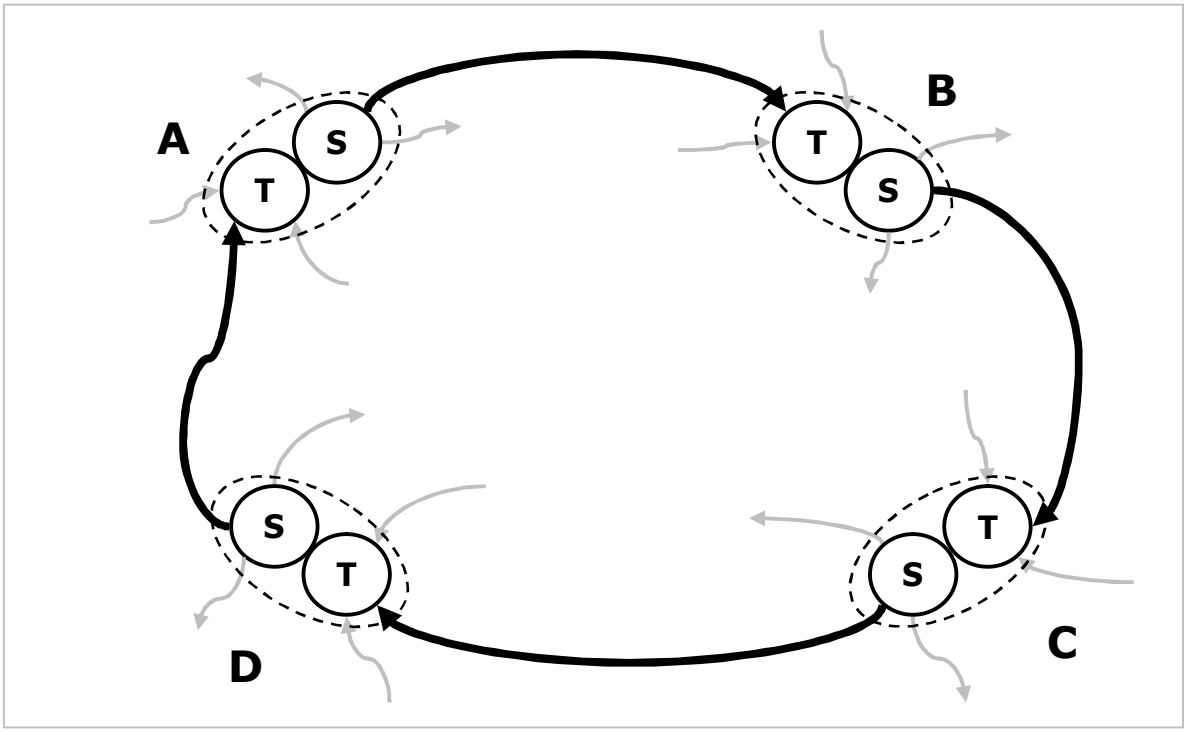


Рис. 32-2 — Раздвоение вершин исходного графа

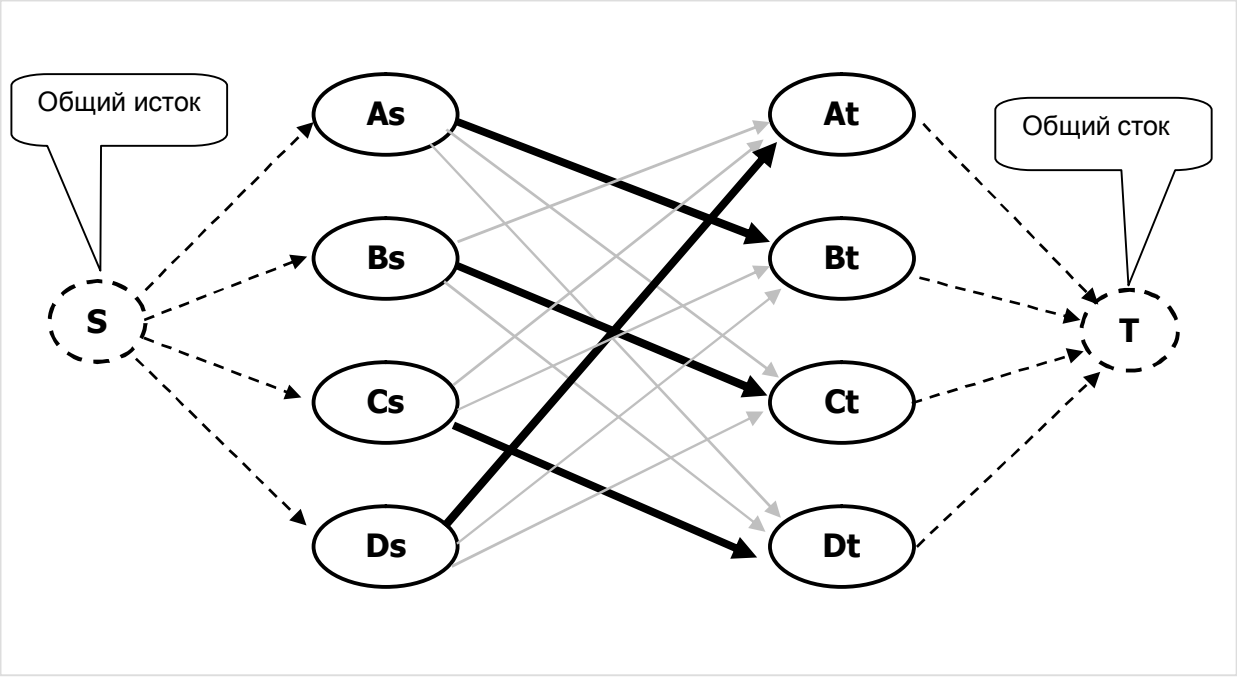


Рис. 32-3 — Эквивалентный двудольный граф

32.3. Не контур, а контуры!

Впрочем, с минимальным потоком я поторопился. Да, венгерский алгоритм даст нам поток минимальной стоимости в двудольном графе. Но, отобразив его на исходный граф, мы можем обнаружить не один, а несколько замкнутых потоков, не связанных между собой (см. рис. 32-4, где граф содержит 8 вершин). И хотя все эти не связанные контуры удовлетворяют одному из требований к контуру Гамильтона — дают совокупность дуг или рёбер *минимальной* длины, — они не отвечают

второму требованию: контур должен быть *единственным*. И для достижения этой цели волшебства уже не достаточно.

Рассмотрим рис. 32-4, где показан случай, когда поиск минимального потока дал два несвязанных контура (эти контуры ещё называют *факторами*). Можно ли объединить эти факторы в один? Разорвём (удалим) дугу одного из контуров, и тогда после повторного поиска минимального потока, инцидентные ему вершины поневоле соединятся в контуры как-то иначе, и вероятность получения *единственного* фактора возрастёт. Эта вероятность повысится ещё больше, если у какой либо вершины разорвать *все* дуги, ведущие в вершины *того же* текущего контура, — тем гарантируется (при повторном поиске) её присоединение к другому контуру. Так, например, для вершины *B* можно разорвать дуги *B-A*, *B-C* и *B-D*, и тогда вершина *B* перекочует в другой контур (рис. 32-4). Разумеется, что применение этой процедуры только к одной вершине не гарантирует получения единственного фактора, и нам придётся поочерёдно применить её к нескольким вершинам. Но, и получив единственный фактор, мы не будем уверены в том, что он минимален, — подобных контуров может быть много, и надо перепробовать отключения всех вершин графа. К чему ведёт всё это?

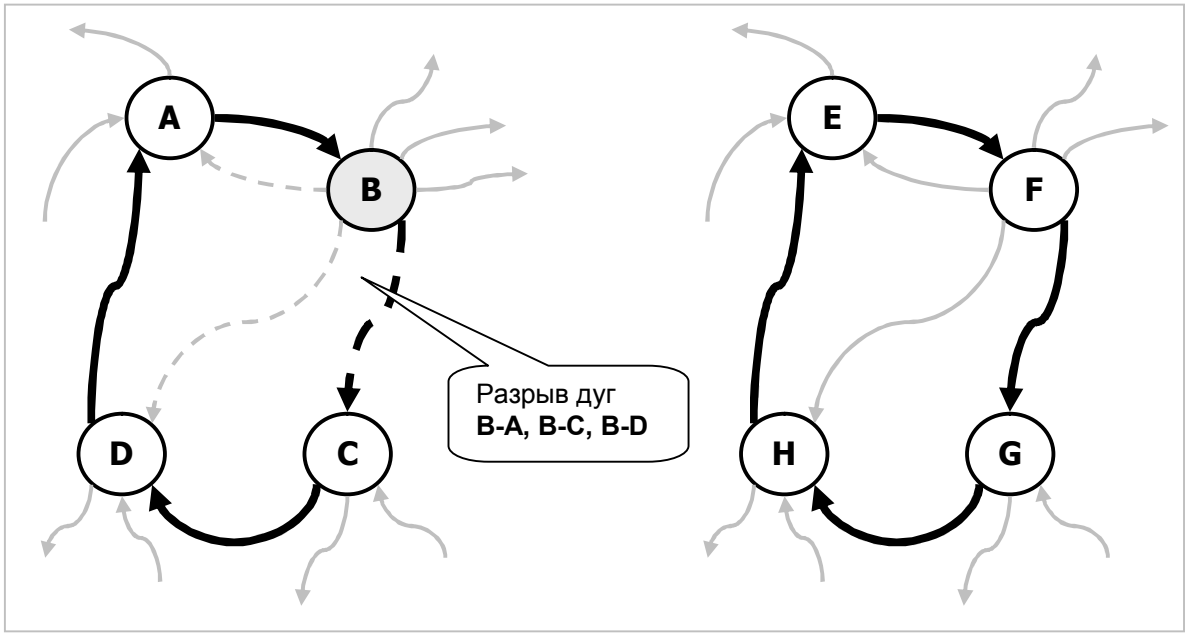


Рис. 32-4 — Минимальный поток, разбитый на два контура (фактора), пунктиром показаны разрываемые дуги для вершины *B*

Пусть первый поиск минимального потока дал нам несколько не связанных факторов. Будем поочерёдно применять выше описанную процедуру разрыва дуг ко всем вершинам графа. Разорвав дуги первой из вершин, найдём соответствующую этому состоянию дуг новую конфигурацию *факторов*. Если фактор (контур) оказался единственным, значит, получен кандидат на гамильтонов контур минимальной стоимости, — запомним его и его стоимость. Если же в полученной конфигурации оказалось несколько факторов, но стоимость соответствующего ей потока оказалась меньше текущего минимума, то с этой конфигурацией можно работать дальше и её следует как-то запомнить. Этим

заканчиваем обработку выбранной вершины, восстанавливаем состояние её дуг, и берём следующую вершину графа. Таким образом, обработка каждой вершины даёт один из двух результатов:

- получена конфигурация потока с единственным фактором — *очередное* решение; если его стоимость оказалась ниже текущего минимума, то это решение и его стоимость запоминаются в качестве лучшего;
- получена конфигурация потока с несколькими факторами — это *промежуточное* решение; если его стоимость оказалась ниже текущего минимума, то сохраняется шанс на улучшение итогового решения, и тогда эту конфигурацию вместе с состоянием дуг сохраняем в приоритетной очереди для дальнейшей обработки.

Проблема является тогда, когда решение с единственным фактором находится не сразу, или когда его начальная стоимость далека от итоговой. Тогда порождается много промежуточных решений, и очередь кандидатов на дальнейшую обработку растёт с геометрической крутизной. В конечном счёте, погоня за точным решением может привести к неприемлемым затратам как памяти, так и времени. И потому далее рассмотрим меры, которые сократят объём вычислений.

32.4. Минимальный фактор

Первая мера состоит в том, чтобы обрабатывать не все, а только те вершины графа, что образуют кратчайший фактор (или один из кратчайших, если таких факторов несколько). Под кратчайшим здесь понимаем фактор, содержащий наименьшее количество вершин. Практика показала, что тем самым стоимость окончательного решения ухудшается очень редко и незначительно, а затраты времени и памяти сокращаются кардинально. Технически выделить минимальный фактор не сложно в ходе подсчёта факторов.

32.5. Уточнённый прогноз стоимости

Вторая мера заключается в более точной оценке прогнозируемой стоимости, достигаемой при последующей обработке промежуточного решения. В качестве этой нижней границы проще всего взять найденную стоимость потока в несвязанных контурах. Однако эта оценка слишком оптимистична, — реальная стоимость единственного контура будет несколько выше, поскольку объединение факторов сопряжено с повышением стоимости потока (и уж точно не с её понижением). Получив более точную оценку нижней границы, мы тем самым (после сравнения её с текущим минимумом) отбросим значительно больше кандидатов на последующую обработку, и установим более точную очерёдность обработки этих кандидатов (самые перспективные обрабатываются раньше).

Базой уточнённого прогноза будет всё та же стоимость потока в несвязных контурах, обозначим её как $C0$, — она находится после отключения части дуг некоторой вершины. Поскольку в последующем эти факторы будут соединены

другими дугами, нижняя граница приращения стоимости этого соединения (*C1*) может быть найдена поиском потока минимальной стоимости на вспомогательном полном графе, полученном *стягиванием факторов* в вершины. Эти вершины соединяются дугами, длины (стоимости) которых выбираются так, чтобы соединить вершины кратчайшими путями из всех возможных. Для этого берутся остаточные (редуцированные) стоимости дуг, полученные после решения задачи о минимальном паросочетании венгерским методом, — их подсчёт уже предусмотрен в нашем венгерском алгоритме. Итак, стянув факторы в вершины и соединив их дугами минимальной (остаточной) стоимости, мы получим некий полный граф. Однако перед вычислением в этом вспомогательном графе минимального потока необходима ещё одна операция, — *компрессия дуг*. Она нужна для соблюдения правила треугольника и состоит в том, чтобы заменить стоимость прямых дуг стоимостью соответствующих кратчайших путей между вершинами, если стоимость последних окажется ниже.

И вот уже на этом полном графе с компрессированными дугами мы вновь находим минимальный поток всё тем же венгерским методом, его цена даст приращение прогноза *C1*, а в целом прогноз составит сумму *C0 + C1*. Если этот второй поток будет образован одним фактором, то процесс вычисления прогноза завершится. Если же факторов опять окажется больше одного, то всё предыдущее повторим уже для вновь полученного графа: стянем факторы в вершины, построим полный граф, выполним компрессию дуг, вычислим стоимость потока *C2* и добавим её к оценке. В худшем случае этот процесс закончится за время, пропорциональное логарифму от количества вершин, а реально гораздо быстрее. На первый взгляд алгоритм кажется громоздким и затратным, на деле он организуется рекурсивно, и не так уж сложен в реализации. В дальнейшем мы вернёмся к нему, рассмотрев числовой пример в пункте 32.9. Забегая вперёд, отметим, что эффект от этой меры весьма заметен на неориентированных графах и симметричных орграфах.

32.6. Общее описание основного алгоритма

После рассмотрения основных идей будущего метода поиска минимального гамильтонова контура (назовём этот метод *основным*), сформулируем его в целом. Для упрощения формулировок вообразим объект, способный хранить промежуточные решения. Положим, что в таком объекте сохраняется следующая информация:

- прогнозируемая стоимость решения;
- текущее множество отключенных дуг;
- множество вершин кратчайшего фактора.

Учредим также *приоритетную* очередь, в которой выше названные объекты с промежуточными решениями будут храниться в порядке не убывания их прогнозируемой стоимости. Тогда алгоритм поиска минимального гамильтонова контура представится так (все действия выполняются на двудольном графе):

- Назначить заведомо большую стоимость текущего лучшего решения.
- Создать объект, хранящий фиктивное промежуточное решение, прогнозируемая стоимость которого равна нулю, множество отключенных дуг пусто, а множество вершин кратчайшего фактора содержит одну (любую) вершину. Поместить этот объект в очередь.
- Пока приоритетная очередь не пуста:
    - Выбрать первый объект, хранящий наименее дорогое промежуточное решение.
    - Если его прогнозируемая стоимость меньше текущего решения, то обработать этот объект следующим образом:
        - Отключить дуги, указанные в множестве отключенных дуг.
        - Для каждой вершины кратчайшего фактора:
        - Отключить дуги, ведущие из этой вершины внутрь фактора.
        - Найти минимальный поток и его стоимость (окончательную или прогнозируемую), а также количество факторов и кратчайший фактор.
        - Если полученная стоимость оказалась меньше текущего минимума, то если количество факторов равно единице, то запомнить это решение в качестве лучшего, а иначе сохранить прогноз стоимости, множество отключенных дуг и кратчайший фактор в объекте хранения и поместить этот объект в приоритетную очередь.
        - Подключить дуги, ведущие из текущей вершины внутрь контура (вернуть состояние).
        - Подключить дуги, указанные в множестве отключенных дуг (вернуть состояние).
    - Уничтожить объект, хранящий промежуточное решение.

Итак, мы начинаем с обработки графа, в котором все дуги подключены, активны. Если в этом состоянии двудольного графа циркулирующий поток минимальной стоимости будет образован одним фактором, то решение найдено. Иначе для кратчайшего фактора создаём несколько объектов хранения промежуточных решений, и ставим их в очередь. Отметим, что с того момента, как стоимость лучшего решения окажется ниже прогнозируемой стоимости первого объекта в очереди, все последующие объекты в этой очереди будут пропущены.

Теперь приступим к деталям этого алгоритма, и для начала рассмотрим вспомогательные механизмы и объекты, принимающие на себя часть его функций.

32.7. Вспомогательные объекты

32.7.1. Механизм отключения дуг

Легко заметить, сколь часто мы будем отключать и вновь включать те или иные дуги графа. Отключение дуг (закрытие) эквивалентно их изъятию из графа, оно приводит к тому, что венгерский алгоритм как бы не замечает этих дуг,

игнорирует их, — такое поведение уже предусмотрено в реализации венгерского метода (см. метод **TGraph.MarkMinPairsDicoty**). Состояние дуги определяется значением поля **TLink.mHigh**: ноль соответствует включенной дуге, единица — отключенной.

32.7.2. Объект для хранения промежуточного решения

Этот объект хранит состояние графа в момент получения очередного промежуточного решения, когда количество факторов отлично от единицы. Поскольку это состояние характеризуется главным образом совокупностью закрытых и закрываемых дуг, объекту дано название **TClosed**. Назначение полей объекта таково:

Поле **mCost** хранит прогнозируемую стоимость потенциального решения, оно определяет положение элемента в приоритетной очереди так, что наименее дорогое и перспективное решение оказывается в начале очереди.

Уникальное поле **mId** необходимо для различения объектов с одинаковой прогнозируемой стоимостью, а иначе дубликаты отбрасывались бы.

Поле **mClosed** содержит множество дуг, которые были закрыты на момент получения решения, — они останутся закрытыми и в продолжение решения.

Поле **mFactors** по идее могло бы хранить множество вершин кратчайшего фактора (как было сказано выше), но технически удобней хранить здесь множество множеств дуг, которые будут закрываться в продолжение решения. Таким образом, каждой вершине кратчайшего фактора здесь соответствует одно множество закрываемых дуг, а количество этих множеств равно количеству вершин в кратчайшем факторе.

Листинг 32-1 — Объект, хранящий промежуточное решение

```
type  TClosed = class (TItem)
        mCost: integer;  // стоимость потока
        mId: integer;    // уникальный идентификатор
        mClosed: TSet;   // множество закрытых дуг (клапанов)
        mFactors: TSet;  // множества отключаемых дуг кратчайшего фактора
        constructor Create(aCost: Integer; aClosed: TSet);
        destructor Destroy;  override;
        function Compare(arg: TItem): TCompare; override;
        procedure InsertLinksSet(aSet: TSet);
        procedure GatesOn;
        procedure GatesOff;
        procedure Print(var aFile: TextFile); override;
      end;

var ClosedId : integer; // уникальный идентификатор элемента очереди
                        // наращивается в конструкторе объекта

constructor TClosed.Create(aCost: Integer; aClosed: TSet);
begin
  inherited Create;
```

```
  Inc(ClosedId);     // наращиваем уникальный идентификатор
  mId:= ClosedId;    // и заносим в поле объекта
  mCost:= aCost;     // стоимость потока на момент создания объекта
  // Создаём и копируем множество закрытых дуг из параметра
  mClosed:= CreateSet;
  if Assigned(aClosed) then mClosed.CopyItems(aClosed);
  // Заготавливаем множество множеств закрываемых дуг
  // (по одному множеству для каждой вершины фактора).
  // В это множество будут вставляться множества закрываемых дуг
  // процедурой TClosed.InsertLinksSet
  mFactors:= CreateSet;
end;

destructor TClosed.Destroy;
begin
  mFactors.ClrAndDestroy;  // уничтожаются множества дуг
  mFactors.Free;
  mClosed.Free;
  inherited;
end;

// Добавляет очередное множество отключаемых дуг aSet

procedure TClosed.InsertLinksSet(aSet: TSet);
begin
  mFactors.Insert(aSet)
end;

// Метод сравнения выстраивает объекты в приоритетной очереди
// в порядке не убывания стоимости,
// а при равной стоимости - в порядке вставки
// путём сравнения уникальных идентификаторов

function TClosed.Compare(arg: TItem): TCompare;
begin
  if mCost < (arg as TClosed).mCost
    then Result:= cmpLess
  else if mCost > (arg as TClosed).mCost
    then Result:= cmpGreate
  else { при равенстве стоимостей сравниваем уникальные идентификаторы }
    if mId < (arg as TClosed).mId
      then Result:= cmpLess
    else if mId > (arg as TClosed).mId
      then Result:= cmpGreate
    else Result:= cmpEq
end;

// Отключает дуги, содержащиеся в mClosed

procedure TClosed.GatesOff;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // дуга отключается единицей
      Link:= GetNext as TLink;
    end;
  end;
end;
```

```
// Включает дуги, содержащиеся в mClosed

procedure TClosed.GatesOn;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0; // дуга подключается нулём
      Link:= GetNext as TLink;
    end;
  end;
end;
```

32.7.3. Вспомогательный двудольный граф и его вершины

Этот вспомогательный граф, как сказано выше, используется для поиска замкнутого потока минимальной стоимости венгерским методом. Граф несёт основную алгоритмическую нагрузку создаваемого метода. Но сначала ознакомимся с типом его вершин **THamNode**.

Этот тип (листинг 32-2) построен на основе **TNode**, стало быть, обладает всеми его полями и методами, но вдобавок содержит и дополнительные, а именно:

mLeft — признак принадлежности вершины к левой или правой доле графа;

mNode — ссылка на родовую вершину исходного графа; две вершины двудольного графа — левая и правая — ссылаются на общую родовую вершину;

mBack — поле для взаимных ссылок пары вершин: левая ссылается на правую, а правая на левую; используется для быстрого обхода контуров.

Унаследованные от **TNode** поля используются следующим образом:

mValue — содержит числовую метку фактора (контура); все вершины одного фактора метятся одним числом;

mLink — содержит линк текущего паросочетания; эти линки указывают направление потока, полученного венгерским методом (применяется только в левых вершинах двудольного графа).

Листинг 32-2 — Описание типа вершин двудольного графа

```
type  THamNode = class (TNode)
        mLeft : boolean;     // признак левой вершины (TRUE)
        mNode : TNode;       // родовая вершина
        mBack : THamNode;    // ссылка справа налево и слева направо
        constructor Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
        function  Compare(arg: TItem): TCompare; override;
        function  GetName: string;  override;
        function  NextCircNode: THamNode;
        function  GenFactorsLinks: TSet;
      end;

constructor THamNode.Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
begin
  inherited Create(0, aOwner);
  mNode:= aNode;     // родовая вершина
  mLeft:= aLeft;     // сторона: TRUE - Left, FALSE - Right
end;

function  THamNode.GetName: string;
begin
  // Сторона + имя родовой вершины
  if mLeft then Result:= 'L.' else Result:= 'R.';
  Result:= Result + mNode.GetName;
end;

// Генерирует множество не отключенных дуг,
// ведущих внутрь текущего фактора (контура)

function THamNode.GenFactorsLinks: TSet;
var Link: TLink;
begin
  Result:= CreateSet;
  Link:= OutLinkFirst;
  while Assigned(Link) do begin
    // Если дуга ведёт внутрь текущего контура (фактора)
    // и она не отключена (mHigh = 0), то сохраняем в результате
    with Link do
    if ((mDest as THamNode).mBack.mValue = Self.mValue) and (mHigh = 0)
      then Result.Insert(Link);  // запомнить в множестве отключаемых
    Link:= OutLinkNext;
  end;
end;

// Возвращает следующую левую вершину текущего контура
// в соответствии с текущим потоком

function THamNode.NextCircNode: THamNode;
begin
  Result:= (mLink.mDest as THamNode).mBack;
end;

// При вставке вершины сортируются по алфавиту

function  THamNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as THamNode).GetName
    then Result:= cmpLess
```

```
  else if GetName > (arg as THamNode).GetName
    then Result:= cmpGreate
end;
```

Теперь рассмотрим основной объект — двудольный граф **THamGraph**, он основан на типе **TGraph** и конструируется на основе родового графа, указанного через параметр конструктора. Двудольный граф ориентирован слева направо.

Основной метод графа — **CalcCostAndFactors** — находит минимальный поток (с учётом текущего состояния дуг), количество факторов, одну из вершин кратчайшего фактора, а также прогнозируемую или конечную стоимость потока (конечная стоимость получается при единственном факторе).

Листинг 32-3 — Двудольный граф для поиска контура Гамильтона

```
type  THamGraph = class (TGraph)
        mGraph: TGraph;   // исходный (родовой) граф
        constructor Create(aGraph : TGraph);
      protected
        function Check: boolean;
        function CalcFactors(var aStart: THamNode): integer;
        procedure SaveBest(aBuf: TBuffer);
        function CalcCostAndFactors(
                      var aFactors: integer; //кол-во факторов
                      var aStart: THamNode  // стартовая вершина
                      ): integer;           // стоимость потока
        function Shrink(aFact: integer): THugeGraph;
      end;

constructor THamGraph.Create(aGraph : TGraph);
var Node: TNode;
    HamL, HamR: THamNode;
    Link : TLink;
begin
  inherited Create('Hamilton',  // имя
                   true,        // дуги ориентированы
                   false,       // вершины не нагружены
                   true);       // дуги нагружены
  mGraph:= aGraph; // ссылка на исходный граф

  // На базе вершин исходного графа создаём пары специальных вершин:
  Node:= aGraph.NodeFirst;
  while Assigned(Node) do begin
    // Создаём пару вершин
    HamL:= THamNode.Create(Self, Node, True);  // исток
    HamR:= THamNode.Create(Self, Node, False); // сток
    // Формируем взаимные ссылки для ускорения доступа:
    HamL.mBack:= HamR;
    HamR.mBack:= HamL;
    // и вставляем вершины в этот граф
    mNodes.Insert(HamL);
    mNodes.Insert(HamR);
    Node:= aGraph.NodeNext;
  end;

  // Формируем дуги, ведущие из левой доли в правую:
  HamL:= NodeFirst as THamNode;
  while Assigned(HamL) do begin
```

```
    // Это левая вершина?
    if HamL.mLeft then  begin
      // Да, обрабатываем её
      PosPush;  // сохраним позицию перебора вершин
      // Перебираем исходящие дуги в вершине исходного графа
      Link:= HamL.mNode.OutLinkFirst;
      while Assigned(Link) do begin
        Node:= Link.mDest;  // целевая вершина исходного графа
        // Ищем соответствующую ей вершину в правой доле:
        HamR:= NodeFirst as THamNode;
        while Assigned(HamR) do begin
          with HamR do if not mLeft and (mNode = Node) then Break;
          HamR:= NodeNext  as THamNode;
        end;
        // Здесь HamR соответствует целевой вершине исходного графа
        // Создаём дугу из левой доли в правую (той же стоимости)
        HamL.MakeLink(HamR, Link.mValue);
        // Переход к следующей дуге исходного графа
        Link:= HamL.mNode.OutLinkNext;
      end; // while
      PosPop;  // восстановим позицию перебора вершин
    end; // if
    HamL:= NodeNext as THamNode;
  end; // while
end;

// Поиск и пометка в поле mValue всех факторов (контуров)
// Возвращает количество факторов.
// Принадлежность вершины к фактору определяется полем mValue

function THamGraph.CalcFactors(var aStart: THamNode): integer;
var NodesCnt: integer;  // счётчик оставшихся необработанных вершин
    Start, Left : THamNode; // стартовая и текущая вершины
    CircLen: integer;   // длина очередного фактора
    MinLen : integer;   // длина кратчайшего фактора
    Link : TLink;
begin
  Result:= 0;          // количество факторов
  MinLen:= MaxInt;     // длина кратчайшего фактора
  // количество вершин исходного графа:
  NodesCnt:= mGraph.mNodes.GetCount;
  // очистка полей mValue двудольного графа:
  Start:= NodeFirst as THamNode;
  while Assigned(Start) do begin
    Start.mValue:= 0;
    Start:= NodeNext as THamNode;
  end;
  // Пока не обработаны все вершины графа (исходного)
  while NodesCnt>0 do begin
    Inc(Result);  // Result= 1, 2, .. - цвет очередного контура
    CircLen:=0;   // длина текущего фактора (контура)
    // Ищем первую неокрашенную вершину в левой доле:
    Start:= NodeFirst as THamNode;
    while Assigned(Start) do begin
      with Start do if mLeft and (mValue = 0) then Break;
      Start:= NodeNext as THamNode;
    end;
    // Начиная с вершины Start обходим двудольный граф
    // отмечая цветом Result очередной замкнутый контур:
    // => вправо - по направлению текущего потока;
    // <= влево  - по обратным ссылкам mBack
    // пока не вернёмся к исходной вершине
```

```
    Left:= Start;
    repeat
      Dec(NodesCnt);         // счётчик оставшихся вершин
      Inc(CircLen);          // длина этого фактора (контура)
      Left.mValue:= Result; // метим контур текущим цветом
      // Left.mLink -- дуга из вершины Left в вершину Right
      // через которую течёт ненулевой поток (чёрный линк паросочетания)
      Link:= Left.mLink;
      // переход в левую долю вдоль потока:
      Left:= (Link.mDest as THamNode).mBack;
    until Left = Start;
    // После обхода контура запоминаем кратчайший:
    if CircLen < MinLen then begin
      MinLen:= CircLen;    // длина фактора (контура)
      aStart:= Start;      // стартовая вершина фактора (любая из)
    end;
  end; // while NodesCnt >0
end;

// Проверка возможности пропуска через двудольный граф
// насыщенного (максимального) потока
// Возвращает:
// FALSE -- если хотя бы одна из вершин не пропускает ни одной единицы потока
// TRUE  -- если через все вершины проходит хоть одна единица потока
// (Link.mHigh <> 0) -- признак закрытой дуги (заблокированной)

function THamGraph.Check: boolean;
var Node: THamNode;
    Link: TLink;
begin
  Result:= false;
  // Перебор вершин
  Node:= NodeFirst as THamNode;
  while Assigned(Node) do begin
    if Node.mLeft then begin
      // В левой доле исследуем исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.OutLinkNext;
      if not Assigned(Link) then Exit;
    end else begin
      // В правой доле исследуем входящие линки
      Link:= Node.InLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.InLinkNext;
      if not Assigned(Link) then Exit;
    end;
    Node:= NodeNext as THamNode;
  end;
  Result:= true;  // насыщенный поток возможен
end;

// Сохранение текущего потока в качестве наилучшего
// Вызывается при обнаружении очередного лучшего контура

procedure THamGraph.SaveBest(aBuf: TBuffer);
var Start, Node : THamNode;
begin
  aBuf.Clear; // очистка буфера результата
  // Начинаем с первой вершины в левой доле
  Start:= NodeFirst as THamNode;
  while not Start.mLeft do Start:= NodeNext as THamNode;
```

```
  // Обходим контур от Start до Start
  Node:= Start;
  repeat
    aBuf.Put(Node.mNode);
    Node:= Node.NextCircNode;
  until Node = Start;
end;

// Стягивание факторов в новый граф для вычисления прогноза стоимости
// aFact - количество факторов
// Вызывается после построения потока и факторов в последовательности:
// MarkMinPairsDicoty -> CalcFactors -> Shrink

function THamGraph.Shrink(aFact: integer): THugeGraph;
var i: integer;
    Link: TLink;
    N1, N2: TNodeInt;   // вершины стянутого графа помечены числами
    NH1, NH2: THamNode; // вершины данного двудольного графа
    Cost: integer;      // остаточная стоимость дуги
begin
  // Создаём пустой граф:
  Result:= THugeGraph.Create('',  // имя
                         true,    // граф направленный
                         false,   // вершины не нагружены
                         true     // дуги нагружены
                        );
  // Создаём и вставляем вершины, помеченные числами от 1 до aFact
  for i:= 1 to aFact do Result.InsertNode(TNodeInt.Create(i, 0, Result));

  // Соединяем дугами бесконечной длины все вершины созданного графа
  with Result do begin
    N1:= NodeFirst as TNodeInt;
    while Assigned(N1) do begin
      PosPush;
      N2:= NodeFirst as TNodeInt;
      while Assigned(N2) do begin
        if N1 <> N2 then SetLink(N1, N2, MaxInt);
        N2:= NodeNext as TNodeInt;
      end;
      PosPop;
      N1:= NodeNext as TNodeInt;
    end;
  end;

  // Находим минимальные относительные стоимости дуг нового графа
  // Остаточные стоимости дуг в полях TLink.mLow
  // определены при поиске минимального паросочетания (потока)
  // процедурой TGraph.MarkMinPairsDicoty
  NH1:= NodeFirst as THamNode;
  while Assigned(NH1) do begin
    // Обработка вершин левой доли:
    if NH1.mLeft then begin
      // Перебираем дуги вершины:
      Link:= NH1.OutLinkFirst;
      while Assigned(Link) do begin
        NH2:= (Link.mDest as THamNode).mBack;
        // Если линк ведёт в другой фактор (контур), то обрабатываем
        if NH1.mValue <> NH2.mValue then begin
          // Остаточная стоимость дуги после редуцирования
          // (определена процедурой TGraph.MarkMinPairsDicoty)
          Cost:= Link.mLow;
          // Находим линк между вершинами (факторами) в стянутом графе
```

```
          // Здесь поля NH1.mValue, NH2.mValue соответствуют факторам
          // и вершинам стянутого графа
          Link:= Result.GetLinkByNumber(NH1.mValue, NH2.mValue);
          // Назначаем этому линку пониженую (редуцированную) цену
          if Link.mValue > Cost then Link.mValue:= Cost;
        end;
        Link:= NH1.OutLinkNext;
      end;
    end;
    NH1:= NodeNext as THamNode;
  end;
  // Выполняем компрессию дуг, заменяя их длину кратчайшими расстояниями
  Result.Compress;
end;

// Оценка прогнозируемой стоимости и подсчёт количества факторов
// при текущем состоянии клапанов (открытых и закрытых дуг)
// Возвращает:
// Result   -- нижний предел стоимости для данного состояния клапанов mHigh
// aFactors -- количество факторов
// aStart   -- стартовую вершину кратчайшего фактора

function THamGraph.CalcCostAndFactors(
                           var aFactors: integer; //кол-во факторов
                           var aStart: THamNode   // стартовая вершина
                           ): integer;            // стоимость потока

var Level: integer; // уровень рекурсии

  // Рекурсивная процедура стягивания факторов

  procedure Local(aGraph: THamGraph);
  var Shrinked: TGraph; // очередной граф, образованный стягиванием факторов
      HamGr: THamGraph; // и соответствующий ему двудольный граф
      Cost: integer;    // стоимость потока
      Factors: integer; // количество факторов
      Dummy: integer;   // количество обнаруженных пар (не используем)
      Node: THamNode;   // любая вершина кратчайшего фактора
  begin
    Inc(Level); // уровень рекурсии +1

    if Level=1 then begin
      // На первом уровне проверим возможность достижения насыщенного потока
      if not aGraph.Check then begin
        // Насыщенный поток невозможен
        Result:= -1;   Exit;
      end;
    end;

    // Вычисляем стоимость полного потока на двудольном графе
    Cost:= aGraph.MarkMinPairsDicoty(Dummy);

    // Накапливаем результат (прогнозируемую стоимость)
    Inc(Result, Cost);

    // Находим факторы (контуры)
    Factors:= aGraph.CalcFactors(Node);

    // На первом уровне рекурсии запоминаем количество факторов
    // и любую вершину кратчайшего фактора

    if Level=1 then begin aFactors:= Factors; aStart:= Node; end;
```

```
    // Если фактор единственный, то выход с возвращением цены
    if Factors = 1 then Exit;

    // При наличии нескольких факторов (не связанных контуров)
    // стягиваем факторы в вершины, создаём новый полный граф
    // и компрессируем его дуги
    Shrinked:= aGraph.Shrink(Factors);

    // Создаём двудольный граф типа THamGraph
    // и рекурсивно повторяем предыдущие шаги

    HamGr:= THamGraph.Create(Shrinked);
    Local(HamGr);

    // Удаляем вспомогательные графы
    HamGr.Free;
    Shrinked.Free;
    Dec(Level); // уровень рекурсии -1
  end;

begin   { THamGraph.CalcCostAndFactors }
  Result:=0;   aFactors:= 0; aStart:= nil;  Level:= 0;
  Local(Self);
end;
```

Метод компрессии дуг реализован в базовом классе **TGraph** и представлен ниже.

Листинг 32-4 — Метод компрессии дуг

```
// Компрессия дуг: замена длин дуг кратчайшими расстояниями
// Используется для поиска гамильтонова контура

procedure TGraph.Compress;
var  N: TNode;
     L: TLink;
    FL: TFarLink;
begin
  // Строим карту с дальними указателями:
  InitMap_Floyd;
  // Заменяем длины дуг кратчайшими путями:
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор дальних указателей:
    FL:= N.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL) do begin
      if (FL.mNodeFar <> N) then begin
        // Ищем линк из данной вершины N --> mNodeFar
        L:= N.GetLink(FL.mNodeFar);
        // Ели он существует, меняем длину на кратчайшее расстояние
        if Assigned(L) then L.mValue:= FL.mDist;
      end;
      FL:= N.mFarLinks.GetNext as TFarLink;
    end;
    N:= NodeNext;
  end;
  // Удаляем карту:
  DoneMap;
end;
```

Ниже представлен текст основного метода поиска гамильтонова контура, который использует описанные выше объекты. Метод возвращает стоимость контура и соответствующую последовательность вершин, начиная с первой.

Листинг 32-5 — Основной метод поиска гамильтонова контура

```
function TGraph.GenHamilton(var aCost: integer): TBuffer;

var   HamGraph: THamGraph; // вспомогательный двудольный граф
      Que : TSet;          // Приоритетная очередь клапанов (по неубыванию)
      BestCost: integer;   // лучшая цена на текущий момент
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Отключение дуг, переданных в множестве aLinks

  procedure LinksOff(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // отключить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Включение дуг, переданных в множестве aLinks

  procedure LinksOn(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0;  // включить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка объекта с клапанами aClosed
  // (множеством отключенных и множествами отключаемых дуг)

  procedure Handle(aClosed: TClosed);
  var Cost: integer;    // прогноз или окончательная стоимость решения
      Factors: integer; // количество факторов в очередном решении
      Start: THamNode;  // стартовая вершина кратчайшего фактора
      Closed: TClosed;  // новый объект с клапанами
      Links: TSet;      // множество отключенных дуг в текущий момент
      Node: THamNode;
  begin
    // Отключить блокированные дуги объекта
    aClosed.GatesOff;

    // Цикл: поочерёдно отключаем дуги, ведущие внутрь фактора,
    // оцениваем стоимость соответствующих потоков,
    // и ставим в приоритетную очередь элементы TClosed
    // для частичных решений

    // Перебор множеств отключаемых дуг:
    Links:= aClosed.mFactors.GetFirst as TSet;
    while Assigned(Links) do begin
      // Отключаем дуги одной из вершин фактора, ведущие внутрь этого фактора
      LinksOff(Links);
      // При текущем состоянии клапанов определить стоимость потока
```

```
      Cost:= HamGraph.CalcCostAndFactors(Factors, Start);
      // Вновь подключаем дуги вершины, ведущие внутрь фактора
      LinksOn(Links);
      // Если поток существует, и он лучше текущего...
      if (Cost>=0) and (Cost< BestCost) then begin
        // Здесь поток существует и его стоимость ниже текущего минимума.
        // Если количество факторов=1, то запомнить лучший результат.
        if Factors=1 then begin
          BestCost:= Cost;            // лучшая цена
          aCost:= Cost;               // она же как результат
          // сохраняем последовательность вершин контура в буфере результата
          HamGraph.SaveBest(Result);
        end else begin
          // Здесь количество факторов Factors > 1 (частичное решение),
          // то создаём и ставим в очередь множество закрытых клапанов
          Links.Add(aClosed.mClosed); // объединяем Links и mClosed
          Closed:= TClosed.Create(Cost, Links);
          // Обходим кратчайший фактор
          // и сохраняем в объекте Closed отключаемые дуги
          Node:= Start; // Начальная (любая) вершина фактора
          repeat
            // Функция THamNode.GenFactorsLinks создаёт множество дуг
            // ведущих внутрь текущего фактора
            Closed.InsertLinksSet(Node.GenFactorsLinks);
            Node:= Node.NextCircNode; // следующая вершина фактора
          until Node = Start;
          Que.Insert(Closed); // Вставляем объект в приоритетную очередь
        end;
      end;
      // Следующее множество отключаемых дуг:
      Links:= aClosed.mFactors.GetNext as TSet
    end;
    // Вернуть исходное состояние клапанов
    aClosed.GatesOn;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var  Closed: TClosed; // очередное множество закрытых клапанов

begin  { TGraph.GenHamilton }

  Result:= TBuffer.Create;  // создаём пустой буфер для хранения решения
  aCost:= -1;               // стоимость на случай отсутствия решения

  // Создаём вспомогательный двудольный граф HamGraph (Dicotyledonous)
  // с целью определения минимальной стоимости полного потока,
  // а также очередь клапанов Que (элементов типа TGate)

  HamGraph:= THamGraph.Create(Self);
  Que:= CreateSet;  // Приоритетная очередь клапанов по неубыванию стоимости
  BestCost:= MaxInt;// Минимальная стоимость на текущий момент

  // Создаём и вставляем в приоритетную очередь фиктивный элемент,
  // в котором множества закрытых и закрываемых линков пусты
  Closed:= TClosed.Create(0, nil);
  Closed.InsertLinksSet(CreateSet);
  Que.Insert(Closed);

  // Цикл обработки приоритетной очереди:
  while Que.GetCount > 0 do begin
    // Выбрать из очереди объект-множество клапанов
    Closed:= Que.GetFirst as TClosed;
```

```
    // Если его прогнозируемая цена меньше лучшей на данный момент,
    // то обработать это состояние клапанов
    if Closed.mCost < BestCost then Handle(Closed);
    // Удалить объект-множество клапанов
    Que.Delete(Closed);
    Closed.Free;
  end;
  // Очистка памяти:
  Que.Free;       // приоритетная очередь клапанов по неубыванию стоимости
  HamGraph.Free;  // вспомогательный двудольный граф (Dicotyledonous)
end;
```

32.8. Испытания

В ходе испытаний основного метода предстоит оценить его точность (в сравнении с полным перебором), а также время работы в зависимости от размера графа. Объектом расчётов будут случайные графы с дугами, длина которых распределена равномерно в интервале от 1 до 99. Ниже дана программа для сравнения четырёх методов поиска на отдельных случайно генерируемых графах.

Листинг 32-6 — Программа для сравнения методов поиска контуров Гамильтона

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

// Преобразование буфера в строку

function HamToStr(aBuf: TBuffer): string;
var N: TNode;
    i: integer;
begin
  Result:='';
  for i:= 1 to aBuf.GetCount do begin
    N:= aBuf.Get as TNode;
    aBuf.Put(N);
    Result:= Result+ N.GetName;
    if i<aBuf.GetCount then Result:=Result +'-';
  end;
end;

var Gr : TGraph;
    Res : TBuffer;
    Cost: integer;
    Nodes: integer;
    Start: TDateTime;
    Time: integer;
    N : integer;
```

```
begin
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes)<3 then Break;
      {
        Nodes > 0 -- оргграф,  Nodes < 0 -- граф
      }
      Gr:= TGraphChars.GenFull((Nodes>0), 0, 99, Abs(Nodes));
      if Gr.Nodes < 18 then begin
        Writeln('-------------- Full --------------------');
        Start:= Now;
        Res:= Gr.GenHamilton_Full(Cost);
        Time:= MilliSecondsBetween(Start, Now);
        Writeln(HamToStr(Res));
        Res.Free;
        Write('Cost    = ', Cost);
        Writeln('  Time (ms) =', Time:5, #10);
        N:= Cost; // Стоимость образца
      end;
      Writeln('-------------- Main --------------------');
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost(T)= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      if Gr.Nodes >= 18 then N:= Cost; // Стоимость образца

      if Cost>0 then begin
      // Жадный и случайный выбор
      Writeln('-------------- Greed true --------------------');
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, true);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      Writeln('R2= ', 100*Cost/N:6:1);
      // Случайный выбор
      Writeln('-------------- Greed false --------------------');
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, false);
      Time:= MilliSecondsBetween(Start, Now);
      Writeln(HamToStr(Res));
      Res.Free;
      Write('Cost= ', Cost);
      Writeln('  Time (ms) =', Time:5, #10);
      Writeln('R3= ', 100*Cost/N:6:1);
      Writeln('-----------------------------------------------'#10);
      end;
      Gr.Free;
    until false;
end.
```

Более достоверные результаты даст обработка многих графов с накоплением статистики. Следующая программа генерирует и обрабатывает заданное пользователем количество полных графов и вычисляет среднее время их обработки, процент точных результатов, а также среднее и максимальное

отклонения стоимости от идеала. В качестве идеала берётся стоимость, полученная либо полным перебором (для небольших графов), либо основным методом. При задании положительного количества вершин генерируются ориентированные несимметричные графы, отрицательного — не направленные графы (то есть, симметричные орграфы).

***Примечание**:* вызываемый в данной программе метод **GenHamilton_Quick** будет рассмотрен в следующей главе.

Листинг 32-7 — Накопление статистики по пяти методам поиска контуров Гамильтона

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';
{-------------------------------------------------------}

procedure Test(aRetry, aNodes: integer);
var Gr : TGraph;             // исследуемый граф
    Res : TBuffer;           // результат - гамильтонов контур
    Cost, CostFull: integer;          // стоимости
    Sum2, Sum3, Sum4, Sum5: Extended; // суммы относительной стоимости в %
    Max2, Max3, Max4, Max5: Extended; // максимальные отклонения в процентах
    OK2, OK3, OK4, OK5: integer;      // количество совпадений с идеалом

    T1, T2, T3, T4, T5 : integer; // время накопленное
    Start : TDateTime;            // для засечки времени

    Cnt : integer;     // счётчик решений
    Delta : integer;   // абсолютное отклонение
    pc :  extended;    // отношение в %

   // Вывод статистики
  procedure Expo(const aFile: String);
  const CLine='-------------------------------------------------------------';
  var  x1, x2, x3, x4, x5 : Extended;
  begin
    if aFile <> '' then begin
      Assign(Output, aFile);
      if FileExists(aFile)  then Append(Output)   else Rewrite(Output)
    end;
    Writeln(CLine);
    Writeln('Retry=', aRetry:4,  '  Nodes=', aNodes:4);
    Writeln(CLine);
    Writeln('                 Full      Ham      Greed    Random     Quick');
    // Относительная стоимость, %:
    x2:= Sum2/aRetry;
    x3:= Sum3/aRetry;
    x4:= Sum4/aRetry;
    x5:= Sum5/aRetry;
    Write('Ratio(%) = ', '':10, x2:10:1, x3:10:1, x4:10:1);
```

```
    if aNodes > 0 then Writeln else Writeln(x5:10:1);

    // максимальные отклонения в процентах
    Write('MaxR (%) = ', '':10, Max2:10:1, Max3:10:1, Max4:10:1);
    if aNodes > 0 then Writeln else Writeln(Max5:10:1);
    // количество совпадений с идеалом в процентах
    Write('OK   (%) = ', '':10, 100*OK2/aRetry:10:1,
                100*OK3/aRetry:10:1, 100*OK4/aRetry:10:1);
    if aNodes > 0 then Writeln else Writeln(100*OK5/aRetry:10:1);
    // Среднее время, мс
    x1:= T1/aRetry; x2:= T2/aRetry; x3:= T3/aRetry; x4:= T4/aRetry;
    x5:= T5/aRetry;
    Write('Time (ms)= ', x1:10:1, x2:10:1, x3:10:1, x4:10:1);
    if aNodes > 0 then Writeln else Writeln(x5:10:1);

    Writeln(CLine);
    if aFile <> '' then begin
      Close(Output);  Assign(Output, '');    Rewrite(Output);
    end;
  end;

begin  { Test }

    // суммы относительной стоимости в %
    Sum2:= 0; Sum3:= 0; Sum4:= 0; Sum5:=0;
    // максимальные отношение в процентах
    Max2:= 0; Max3:=0; Max4:=0; Max5:=0;
    // количество совпадений с идеалом
    OK2:=0; OK3:=0; OK4:=0; OK5:=0;
    // время накопленное
    T1:=0; T2:=0; T3:=0; T4:=0; T5:=0;

    for Cnt:= 1 to aRetry do begin
      // aNodes>0 - орграф,  aNodes<0 - симметричный граф
      Gr:= THugeGraph.GenFull((aNodes>0), 0, 99, Abs(aNodes));

      if Abs(aNodes) > 15 then CostFull:=-1
      else begin
        {-------- Full -------}
        Start:= Now;
        Res:= Gr.GenHamilton_Full(CostFull);
        Inc(T1, MilliSecondsBetween(Start, Now));
        Res.Free;
      end;

      {-------- Main -------}
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Inc(T2, MilliSecondsBetween(Start, Now));
      Res.Free;
      if CostFull < 0 then CostFull:= Cost;
      Delta:= Cost - CostFull;
      if Delta=0 then Write('.') else Write('*');
      if Delta = 0 then Inc(OK2);
      pc:= 100 * Cost/CostFull;
      Sum2:= Sum2 + pc;
      if Max2 < pc then Max2:= pc;

      {-------- Greed -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, true);
      Inc(T3, MilliSecondsBetween(Start, Now));
```

```
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK3);
      pc:= 100 * Cost/CostFull;
      Sum3:= Sum3 + pc;
      if Max3 < pc then Max3:= pc;

      {-------- Random -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Greed(Cost, false);
      Inc(T4, MilliSecondsBetween(Start, Now));
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK4);
      pc:= 100 * Cost/CostFull;
      Sum4:= Sum4 + pc;
      if Max4 < pc then Max4:= pc;

      {-------- Quick -------}
      Start:= Now;
      Res:= Gr.GenHamilton_Quick(Cost);
      Inc(T5, MilliSecondsBetween(Start, Now));
      Res.Free;
      Delta:= Cost - CostFull;
      if Delta = 0 then Inc(OK5);
      pc:= 100 * Cost/CostFull;
      Sum5:= Sum5 + pc;
      if Max5 < pc then Max5:= pc;

      Gr.Free;
    end; // for
    Writeln(#7);
    // Вывод статистики
    Expo('');
    Expo('Out.txt');
end;
{-------------------------------------------------------}
var Nodes: integer;
    Retry : integer;

begin
    Write('Retry= '); Readln(Retry);
    if Retry <= 0 then Exit;
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes) < 4 then Break;
      Test(Retry, Nodes);
    until false;
end.
```

Ниже даны результаты обработки после испытаний по 200 *графов* с количеством вершин от 11 до 14, и по 500 *орграфов* с тем же количеством вершин. Погрешность вычислений в сравнении с полным перебором дана в табл. 32-1.

Табл. 32-1 — Погрешность вычисления стоимости контура в сравнении с полным перебором, %

Алгоритм	Средняя погрешность		Максимальная погрешность	
	Орграфы	Графы	Орграфы	Графы
Основной алгоритм	менее 0,1	0,1	5,6	9,2
Жадный алгоритм	64,6	36,2	233,0	132,0
Случайный выбор	294,6	224,2	913,0	517,0

Отметим высокую точность основного алгоритма, а также существенную разницу результатов для орграфов и ненаправленных графов (симметричных орграфов).

Среднее время обработки одного графа дано в табл. 32-2 (для жадного и случайного алгоритмов времена не показаны ввиду их ничтожной малости в сравнении с другими алгоритмами).

Табл. 32-2 — Среднее время обработки одного графа, мс

Количество вершин	Полный перебор		Основной алгоритм	
	Орграфы	Графы	Орграфы	Графы
11	9,7	21,0	1,3	3,7
12	31,8	74,0	1,6	6,1
13	95,2	265,0	2,2	8,9
14	297,0	920,0	3,1	12,9

Зависимость среднего времени обработки одной вершины от количества вершин показана на следующих рисунках.

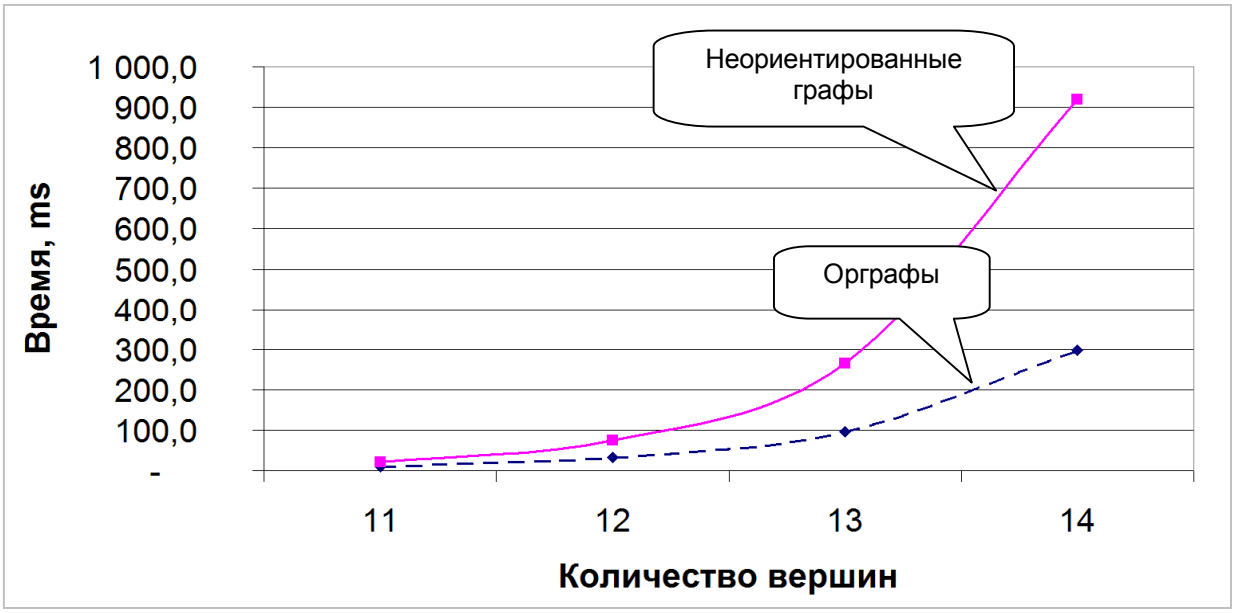


Рис. 32-5 — Зависимость среднего времени *полного перебора* от количества вершин

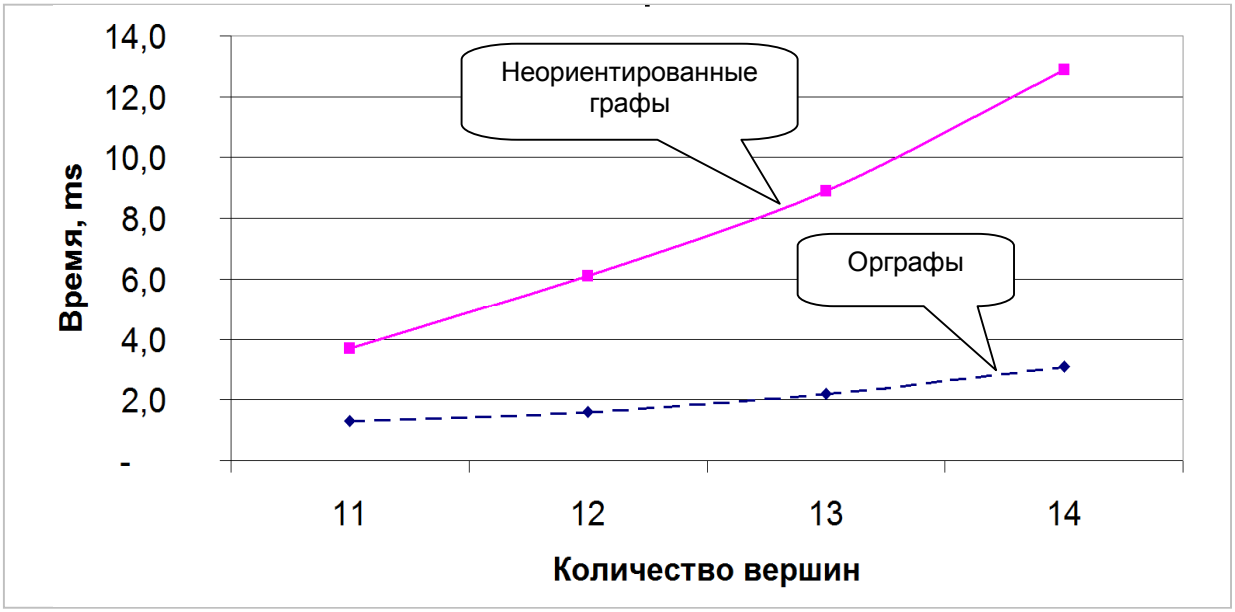


Рис. 32-6 — Зависимость среднего времени работы *основного алгоритма* от количества вершин

Из приведенных выше результатов напрашиваются следующие выводы:

- Основной алгоритм обладает в целом и высокой точностью, и высокой скоростью работы.
- Точность и скорость основного алгоритма заметно ниже при обработке неориентированных графов (симметричных орграфов).
- Жадный алгоритм много уступает в точности основному, но ввиду непревзойдённой скорости его применение иногда может быть оправдано.

32.9. Разбор уточнённого прогноза

Напоследок рассмотрим числовой пример, подтверждающий силу уточнённого прогноза, и воспользуемся для этого симметричным графом, представленным в табл. 32-3.

Табл. 32-3 — Исходный симметричный граф

Источники дуг	Приёмники дуг									
	A	B	C	D	E	F	G	H	I	J
A	*	32	41	22	20	57	54	32	22	45
B	32	*	22	50	42	51	61	20	54	51
C	41	22	*	63	41	30	45	10	60	36
D	22	50	63	*	36	78	72	54	20	64
E	20	42	41	36	*	45	36	32	22	28
F	57	51	30	78	45	*	22	32	67	20
G	54	61	45	72	36	22	*	41	57	10
H	32	20	10	54	32	32	41	*	50	32
I	22	54	60	20	22	67	57	50	*	50
J	45	51	36	64	28	20	10	32	50	*

Описанные далее действия относятся главным образом к методу **`THamGraph.CalcCostAndFactors`**.

В исходном состоянии все дуги графа открыты. Венгерский алгоритм даёт здесь полный замкнутый поток минимальной стоимости, представленный в табл. 32-4 и на рис. 32-7. Этот поток образован четырьмя факторами, его стоимость (длина) составляет 184 единицы.

Табл. 32-4 — Факторы (4), образующие полный поток минимальной стоимости в исходном графе

№	Фактор (контур)	Обозначение стянутой вершины
1	A → E → A	a1
2	B → H → C → B	b1
3	D → I → D	c1
4	F → J → G → F	d1

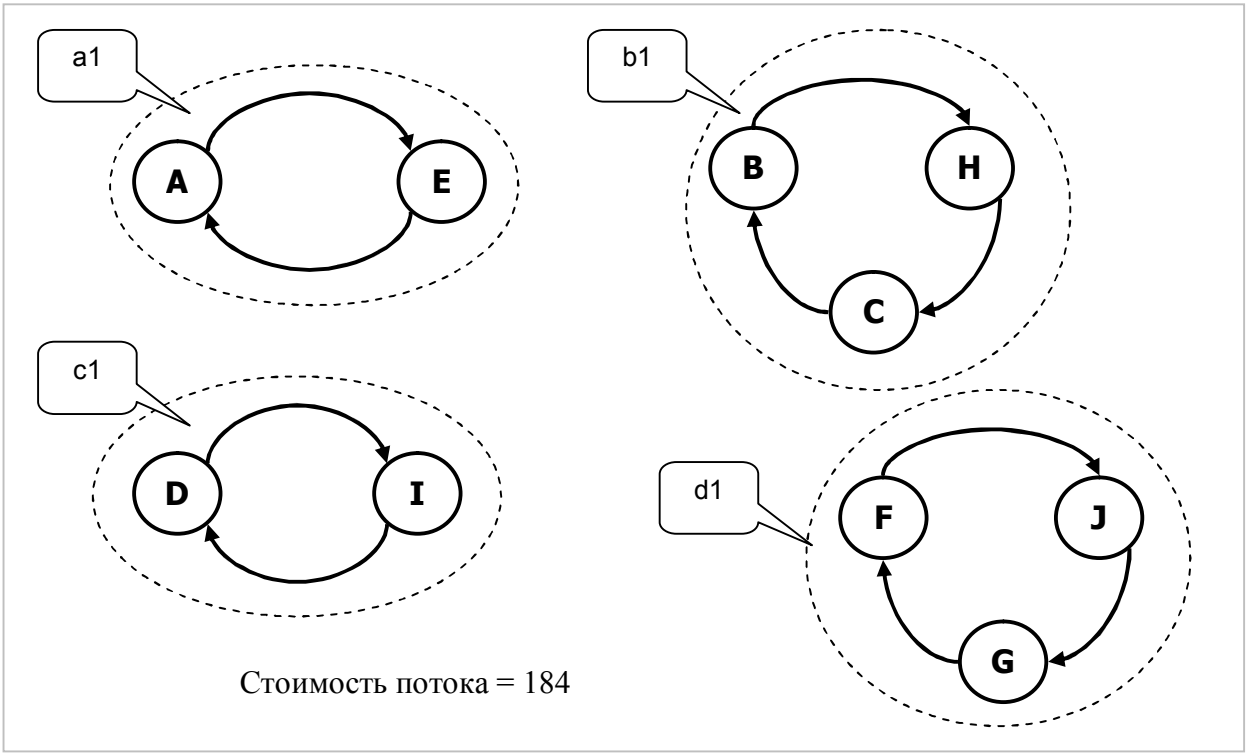


Рис. 32-7 — Факторы (4), образующие полный поток минимальной стоимости

Остаточная (редуцированная) стоимость дуг исходного графа сохранена в полях **`TLink.mLow`** и дана в табл. 1-5. Из этих данных строим полный стянутый граф, где четырём факторам будут сопоставлены вершины **`a1`**, **`b1`**, **`c1`**, **`d1`** (см. рис. 32-8).

Табл. 32-5 — Остаточная (редуцированная стоимость дуг исходного графа

Источники дуг	Приёмники дуг									
	A	B	C	D	E	F	G	H	I	J
A	*	12	31	2	0	37	44	24	2	37
B	0	*	0	18	10	19	39	0	22	31
C	19	0	*	41	19	8	33	0	38	26
D	2	30	53	*	16	58	62	46	0	56
E	0	22	31	16	*	25	26	24	2	20
F	25	19	8	46	13	*	0	12	35	0
G	32	39	33	50	14	0	*	31	35	0
H	12	0	0	34	12	12	31	*	30	24
I	2	34	50	0	2	47	47	42	*	42
J	25	31	26	44	8	0	0	24	30	*

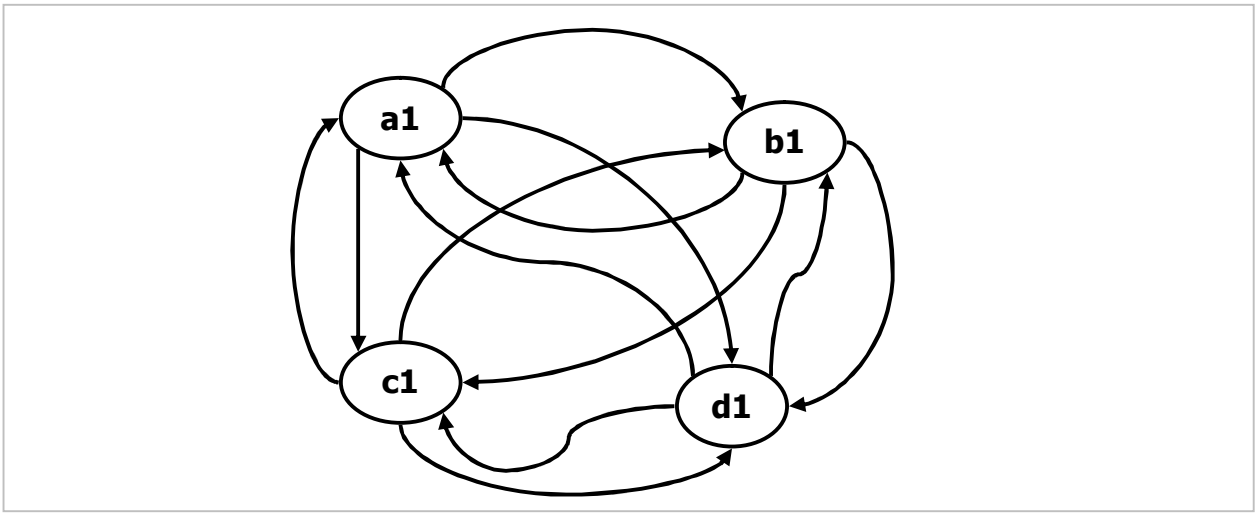


Рис. 32-8 — Полный стянутый граф

Чтобы назначить стоимость дугам стянутого графа (в полях **`TLink.mValue`**), надо пересмотреть все дуги, ведущие из одного подмножества вершин в другое. Так, например, для дуги **a1→ b1** необходимо перебрать дуги, показанные в табл. 32-6, и выбрать среди них кратчайшую (самую дешёвую).

Табл. 32-6 — Перебор дуг, направленных из фактора a1 в фактор b1 (из подмножества A-E в подмножество B-C-H)

Дуга	Остаточная стоимость	Минимум
A → B	12	12
A → C	31	
A → H	24	
E → B	22	
E → C	31	
E → H	24	

Перебрав, таким образом, все сочетания стянутых вершин, мы получим полный стянутый граф, показанный в табл. 32-7.

Табл. 32-7 — Стянутый граф 1-го уровня до компрессии

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	12	2	20
b1	0	*	18	8
c1	2	30	*	42
d1	8	8	30	*

Но прежде, чем применить к этому графу венгерский метод, обеспечим правило треугольника, то есть, проследим за тем, чтобы длины всех дуг в графе оказались не длиннее «окольных путей» между двумя вершинами. Здесь правило нарушено для четырёх дуг (они выделены серым). Например, длина прямой дуги **b1 → c1** составляет 18 единиц, а длина обходного пути **b1 → a1 → c1** равна всего 2 единицам, поэтому меняем здесь 18 на 2. Так же поступаем с остальными дугами (табл. 32-8), после чего получаем сжатый стянутый граф (табл. 32-9).

Табл. 32-8 — Компрессия дуг (замена длины дуги длинной кратчайшего пути между вершинами)

Дуга	Длина дуги	Кратчайший путь	Длина пути
b1 → c1	18	b1 → a1 → c1	2
c1 → b1	30	c1 → a1 → b1	14
c1 → d1	42	c1 → a1 → d1	22
d1 → c1	30	d1 → a1 → c1	10

Табл. 32-9 — Стянутый граф 1-го уровня после компрессии

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	12	2	20
b1	0	*	2	8
c1	2	14	*	22
d1	8	8	10	*

Венгерский алгоритм найдёт в этом графе поток стоимостью 20 единиц, — эта стоимость добавится к прогнозу (184+20). Поток представится двумя факторами, показанными в табл. 32-10 и на рис. 32-9.

Табл. 32-10 — Факторы (2), образующие полный поток минимальной стоимости в стянутом графе 1-го уровня

№	Фактор (контур)	Обозначение стянутой вершины
1	a1 → c1 → a1	a2
2	b1 → d1 → b1	b2

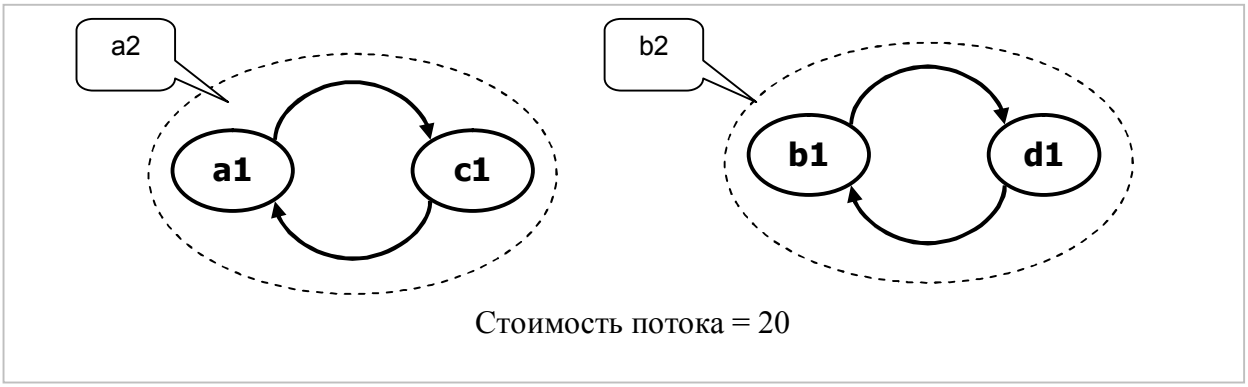


Рис. 32-9 — Факторы (2), полученные в результате поиска полного потока минимальной стоимости в стянутом графе 1-го уровня

По завершении венгерского алгоритма остаточная стоимость дуг представится матрицей в табл. 32-11.

Табл. 32-11 — Остаточная (редуцированная) стоимость дуг в стянутом графе 1-го уровня

Источники дуг	Приёмники дуг			
	a1	b1	c1	d1
a1	*	4	0	12
b1	0	*	0	0
c1	0	4	*	12
d1	8	0	8	*

Поскольку найденный поток вновь образован не одним фактором, продолжим уточнение прогноза, повторив (рекурсивно) выше описанные шаги: стянем факторы в вершины, найдём кратчайшие длины дуг и выполним их компрессию. Результат показан в табл. 32-12, здесь компрессия дуг ничего не изменила, поэтому показана лишь одна таблица.

Табл. 32-12 — Стянутый граф второго уровня (до и после компрессии)

Источники дуг	Приёмники дуг	
	a2	b2
a2	*	4
b2	0	*

Легко догадаться, что венгерский метод найдёт здесь единственный фактор стоимостью 4 единицы, в итоге прогноз стоимости составит `184+20+4=208` единиц. Поскольку дальше уточнять прогноз невозможно, происходит рекурсивный выход из процедуры с уничтожением вспомогательных объектов.

Табл. 32-13 — Фактор, полученный в результате поиска полного потока минимальной стоимости в стянутом графе 2-го уровня

№	Фактор (контур)	Обозначение стянутой вершины
1	a2 → b2 → a2	a3

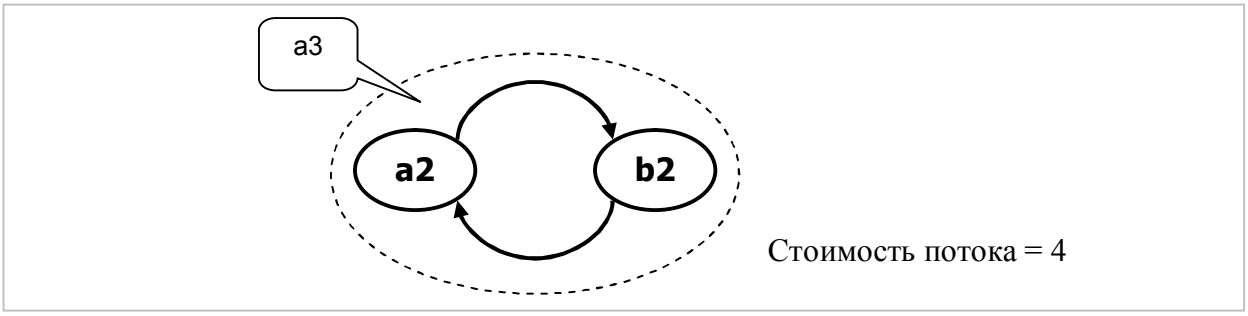


Рис. 32-10 — Единственный фактор, полученный в результате поиска полного потока минимальной стоимости в стянутом графе 2-го уровня

Мы нашли уточнённый прогноз стоимости промежуточного решения в ситуации, когда все вершины графа подключены. Точно так же это делается при последующем отключении некоторых вершин. Проследим за процессом поиска решения, воспользовавшись табл. 32-14. Здесь в первой колонке (шаг) указан номер обрабатываемого промежуточного решения — объекта **`TClosed`**. Начинаем с фиктивного объекта, в котором нет отключенных дуг. В последней колонке указана текущая цена решения, в начальный момент она равна **`MaxInt`** — «бесконечность», что в таблице отмечено прочерком.

Табл. 32-14 — Последовательность промежуточных решений (объектов TClosed) с уточнённым прогнозом

Шаг	Исх. прогноз	Отключенные дуги	Отключа-емые дуги	Состав прогноза	Конечный прогноз	Текущее решение
1	0	нет	нет	184+20+4	208	нет
2	208	нет	A-E	188+24	212	нет
		нет	E-A	188+24	212	нет
3	212	A-E	B-C, B-H	206+7	213	нет
		A-E	C-B, C-H	204+8	212	нет
		A-E	H-B, H-C	208+32	240	нет
4	212	E-A	B-C, B-H	206+12+0	218	нет
		E-A	C-B, C-H	204+12	216	нет
		E-A	H-B, H-C	208+12	220	нет
5	212	A-E, C-B, C-H	B-H	208+16	224	нет
	212	A-E, C-B, C-H	H-B	212+0	212	нет
6	212	A-E, C-B, C-H, H-B	D-I	216	216	216
		A-E, C-B, C-H, H-B	I-D	215+10	225	216
7	213	A-E, B-C, B-H	C-H	212+7	219	216
		A-E, B-C, B-H	H-C	215+1	216	216
8	216	E-A, C-B, C-H	*	*	*	216
9	218	E-A, B-C, B-H	*	*	*	216

Шаг	Исх. прогноз	Отключенные дуги	Отключа-емые дуги	Состав прогноза	Конечный прогноз	Текущее решение
10	**220**	**E-A, H-B, H-C**	*	*	*	**216**
11	**224**	**A-E, C-B, C-H, B-H**	*	*	*	**216**
12	**240**	**A-E, H-B, H-C**	*	*	*	**216**

Итак, при вычислении прогноза для стартового объекта **TClosed** обнаружены (см. выше) четыре фактора, и кратчайший из них: **A** → **E** → **A**. Поэтому при обработке следующего промежуточного решения №2 с исходным прогнозом **208** мы поочерёдно отключаем дуги **A** → **E** и **E** → **A**, получая при этом ещё два промежуточных решения с прогнозами, равными **212** единиц. Оба они не дают конечного решения, поскольку образуются несколькими факторами, и потому ставятся в очередь. Далее (№3 и №4) они выбираются из очереди, и к уже отключенным дугам поочерёдно добавляются дуги кратчайшего фактора **B** → **C** → **H** → **B**. Эти решения вновь не дают единственного фактора, и потому ставятся в приоритетную очередь. Наконец, при обработке объекта №6 и отключении соответствующих дуг, мы получаем единственный фактор стоимостью **216**, — эту конфигурацию потока и его цену запоминаем в качестве текущего решения. При обработке объектов №6 и №7 получаются ещё три промежуточных решения с прогнозами **225**, **219** и **216** единиц, но они уже не ставятся в очередь, поскольку эти прогнозы не меньше цены текущего решения (**216**). Начиная с объекта №8, мы уже не обрабатываем оставшиеся в очереди промежуточные решения, поскольку их прогнозы не меньше текущего решения, — эти объекты просто уничтожаются (отмечены звёздочками).

Итак, решение найдено при обработке 6-го объекта, а всего в очередь попало **12** объектов. Теперь сравним этот результат с упрощённым вариантом, когда прогноз ограничивается лишь первым уровнем рекурсии, — для этого в процедуре **THamGraph.CalcCostAndFactors** организуем досрочный выход, отключив комментарием часть оператора:

```
    // Если фактор единственный, то выход с возвращением цены
    { if Factors = 1 then } Exit;
```

Результат дан в табл. 32-15, где показаны прогноз и конечное решение. Теперь оно получено только на **15**-м шаге, а количество объектов в очереди достигло **33**. Отметим, также, что и после обнаружения решения на шагах с **16** по **26** обрабатываются ещё **11** объектов из очереди, поскольку их прогнозы оказались ниже цены текущего решения. И только, начиная с **27**-го, все последующие объекты игнорируются (отмечены звёздочками).

Табл. 32-15— Последовательность обработки промежуточных решений (объектов TClosed) с упрощённым прогнозом

Шаг	Исх. прогноз	Текущее решение	Шаг	Исх. прогноз	Текущее решение	Шаг	Исх. прогноз	Текущее решение
1	0	нет	12	208	нет	23	215	216
2	184	нет	13	208	нет	24	215	216
3	188	нет	14	208	нет	25	215	216
4	188	нет	15	212	216	26	215	216
5	204	нет	16	212	216	27	219 *	216
6	204	нет	17	212	216	28	220 *	216
7	206	нет	18	212	216	29	220 *	216
8	206	нет	19	212	216	30	220 *	216
9	208	нет	20	212	216	31	222 *	216
10	208	нет	21	212	216	32	226 *	216
11	208	нет	22	212	216	33	226 *	216

Испытания на большом наборе случайных графов показало, что на несимметричных орграфах уточнённый (и усложнённый) прогноз может немного замедлять алгоритм. Зато на ненаправленных графах и симметричных орграфах (как в рассмотренном примере) он даёт ощутимое преимущество.

32.10. Итоги

32.10.1. Замкнутая задача Гамильтона состоят в поиске кратчайшего (или длиннейшего) пути посещения *всех* вершин графа, который начинается и заканчивается в одной и той же (любой) вершине графа.

32.10.2. Так же, как и разомкнутая, замкнутая задача может быть решена полным перебором возможных путей (экспоненциально сложным по времени).

32.10.3. Основной быстрый алгоритм решения замкнутой задачи основан на построении эквивалентного двудольного графа, и применении к нему венгерского алгоритма с целью поиска минимального замкнутого потока.

32.10.4. Поскольку венгерский алгоритм на полном графе может давать несколько замкнутых потоков (факторов), для получения единственного контура необходимо отключать часть дуг графа.

32.10.5. Порядок выбора отключаемых дуг определяет точность и скорость алгоритма. Для сокращения вычислений поочерёдно отключают дуги только тех вершин, что образуют один из кратчайших факторов очередного промежуточного решения.

32.10.6. Скорость вычислений может быть повышена более точным прогнозом конечной стоимости, соответствующей некоторому промежуточному решению. Этот прогноз вычисляется рекурсивно и основан на стягивании факторов в вершины.

32.11. Задания

32.12. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 273
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Глава 33
Гамильтон: комбинации методов

В главах 31 и 32 решены *разомкнутая* и *замкнутая* задачи Гамильтона. Для *разомкнутой* задачи найдено неплохое и быстрое решение — метод штрафования вершин, однако он годен лишь для неориентированных графов, — теперь предстоит найти решение и для орграфов. С другой стороны, основному методу решения *замкнутой* задачи — на основе венгерского алгоритма — трудно даются неориентированные графы. Мы смягчим эту проблему, применив к замкнутой задаче метод штрафования вершин.

Таким образом, «перекрёстное» применение методов, разработанных ранее для замкнутой и разомкнутой задач, даст дополнительный эффект. В следующей таблице показана применимость двух основных методов к различным сочетаниям гамильтоновых задач. Жирной рамкой отмечены сочетания, соответствующие уже решённым задачам. Далее мы сосредоточимся на двух комбинациях, которые отмечены серым цветом.

Табл. 33-1 — Применимость двух основных методов к вариантам гамильтоновых задач

Основной метод	Разомкнутая задача		Замкнутая задача	
	Орграф	Неориентированный граф	Орграф	Неориентированный граф
Метод штрафования вершин	**Не применим**	**Эффективен**	**Не применим**	**Эффективен**
Поиск факторов венгерским методом	**Эффективен**	**Не применим**	**Эффективен**	**Применим, но не вполне эффективен**

33.1. Разомкнутая задача на орграфе

Напомню формулировку разомкнутой задачи. Дан сильно связанный орграф и две его (разные) вершины X и Y. Необходимо найти кратчайший (или длиннейший) путь из вершины X в вершину Y, проходящий через все прочие вершины графа.

Эту задачу легко свести к уже решённой замкнутой задаче на орграфе. Добавим в граф ещё одну вершину Z, и соединим её двумя «бесплатными» дугами с вершинами X и Y так, как показано на рис. 32-1. То есть, длина (стоимость) добавленных дуг принята равной нулю.

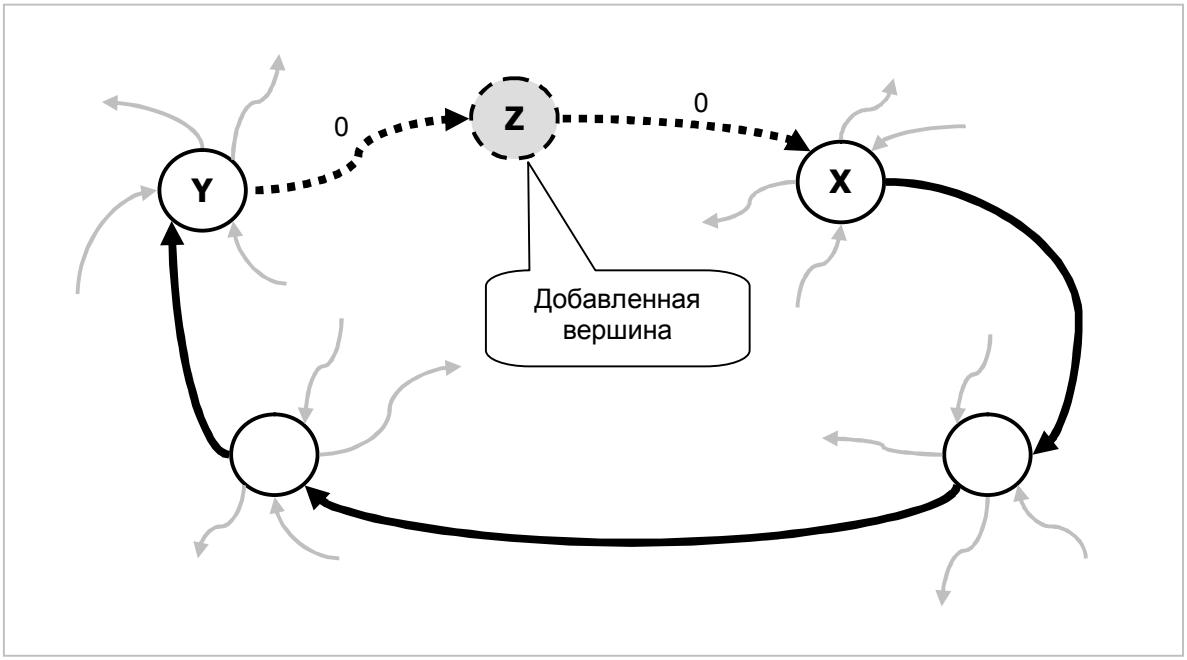


Рис. 33-1 — Вставка вспомогательной вершины Z

Найдём в этом орграфе замкнутый контур Гамильтона уже известным нам методом на основе венгерского алгоритма. Этот контур неминуемо пройдёт через вспомогательные дуги. Если теперь из решения выбросить вспомогательную вершину и её дуги, то останется искомый путь из *X* в *Y* (см. листинг 33-1).

Листинг 33-1 — Метод поиска гамильтонова пути в ориентированном графе

```
function TGraph.GenHamPathDir(aStart, aFin: TNode;
                              var aCost: integer): TBuffer;
var Closer,       // Замыкающая вершина
    Node: TNode;
begin
  Closer:= TNode.Create(0, Self);  // создаём вспомогат. замыкающую вершину
  InsertNode(Closer);              // вставляем её в граф
  Closer.MakeLink(aStart, 0);      // создаём связь Closer -> aStart
  aFin.MakeLink(Closer, 0);        // создаём связь aFin -> Closer
  Result:= GenHamilton(aCost);     // находим контур Гамильтона
  // Удаляем из буфера результата замыкающую вершину
  repeat
    Node:= Result.Get as TNode;
    if Node = Closer then Break;
    Result.Put(Node);
  until false;
  aFin.RemoveLink(Closer);     // удаляем связь aFin -> Closer
  Closer.RemoveLink(aStart);   // удаляем связь Closer -> aStart
  RemoveNode(Closer);          // удаляем замыкающую вершину из графа
  Closer.Free;                 // уничтожаем вершину
end;
```

Излишне объяснять, что в отношении точности и скорости этот алгоритм обладает теми же характеристиками, что и основной алгоритм для замкнутого варианта гамильтоновой задачи (на основе венгерского алгоритма). В следующем листинге представлен универсальный метод решения разомкнутой гамильтоновой задачи, учитывающий род графа (орграф или неориентированный).

Листинг 33-2 — Универсальный метод решения разомкнутой гамильтоновой задачи

```
function TGraph.GenHamPath(aStart,        // начальная вершины
                           aFin: TNode;   // конечная вершины
                           var aCost: integer  // длина (стоимость) цепи
                           ): TBuffer;
begin
  if mDirect
    // Для орграфа используется основной метод поиска гамильтонова контура
    then Result:= GenHamPathDir(aStart, aFin, aCost)
    // Для неориентированного графа - метод штрафования верши
    else Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;
```

33.2. Замкнутая задача на неориентированном графе

Основной метод решения замкнутой задачи на основе венгерского алгоритма в целом справляется с графами обоих типов: и ориентированными, и неориентированными (симметричными). Однако симметричные даются ему куда труднее. Это видно из рис. 33-2: здесь с ростом размерности графа соотношение времён быстро увеличивается.

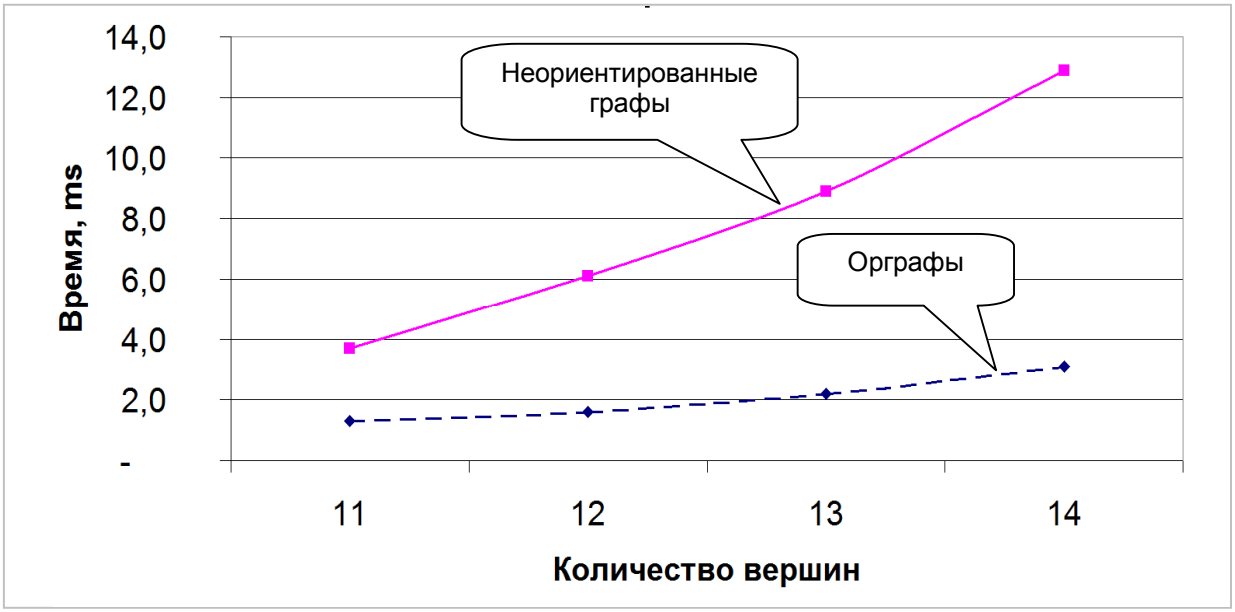


Рис. 33-2— Зависимость среднего времени работы основного метода для решения замкнутой задачи от количества вершин

Причина легко объяснима: обработка симметричных графов порождает много коротких факторов, и соответственно больше промежуточных решений.

Решим эту проблему с привлечением быстрого метода штрафования вершин и некоторых идей, не лишённых здравого смысла. Идеи исходят из того, что замкнутый и разомкнутый варианты задачи отличаются лишь одним ребром. Найдём в графе кратчайшее ребро *X*-*Y*, а затем кратчайший путь из вершины *X* в вершину *Y*. Замкнув затем этот путь кратчайшим ребром, получим контур, часто не слишком уступающий идеальному решению, а порой и совпадающий с ним. Предлагаю вам исследовать этот вариант самостоятельно.

Другая идея (дающая несколько лучший результат) состоит в том, чтобы временно удалить из исходного графа пару рёбер, а заодно и вершину, им инцидентную. Как выбрать эту вершину и рёбра? Для этого пересмотрим все вершины симметричного графа и найдём ту из них, для которой сумма двух кратчайших инцидентных ей рёбер окажется минимальной. Так мы найдем некую вершину *X* и двух её ближайших соседей *Y* и *Z*, — назовём эту троицу вершин *дружной*. Предположим теперь, что нам уже известен кратчайший контур Гамильтона, и случилось так (а вероятность этого не так уж мала), что *дружная* троица выстроилась в этом контуре в том же порядке: *Y-X-Z* или *Z-X-Y*. Здесь мы вправе ожидать, что тот же результат мы получим иным порядком, а именно: найдя кратчайший гамильтонов путь из *Y* в *Z* (исключив при этом вершину *X*), а затем, замкнув путь временно исключённой вершиной и рёбрами. Разумеется, что везёт далеко не всегда, но даже тогда, когда *дружная* троица не содержится в идеальном гамильтоновом контуре, полученный таким образом результат может оказаться неплохим приближением к идеалу. Ниже представлен листинг, реализующий эту идею.

Листинг 33-3 — Ускоренный метод поиска гамильтонова контура с привлечением метода штрафования вершин

```
function TGraph.GenHamilton_Quick(var aCost: integer): TBuffer;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка очередной вершины:
  // возвращаются две ближайшие вершины (aY, aZ)
  // и сумма расстояний

  function FindX(aX : TNode; var aY, aZ : TNode): integer;
  var L: TLink;
      YVal, ZVal : integer; // расстояния к соседям
  begin
    aY:= nil; aZ:= nil;
    YVal:= MaxInt; ZVal:= MaxInt;
    // Перебор исходящих линков:
    L:= aX.OutLinkFirst;
    // Всегда YVal <= ZVal
    while Assigned(L) do begin
      if L.mValue <= YVal then begin
        ZVal:= YVal;
        YVal:= L.mValue;
        aZ:= aY; aY:= L.mDest;
      end else if L.mValue < ZVal then begin
        ZVal:= L.mValue;
        aZ:= L.mDest;
      end;
      L:= aX.OutLinkNext;
    end;
    // Возвращаем сумму расстояний к ближайшим соседям
    Result:= YVal + ZVal;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины с минимальной суммой расстояний
  // к двум ближайшим соседям

  function FindNode(var aX, aY, aZ : TNode): integer;
  var X, Y, Z : TNode;
      Cost, BestCost: integer;
```

```
  begin
    aX:= nil; aY:= nil; aZ:= nil;
    BestCost:= MaxInt;
    X:= NodeFirst;
    while Assigned(X) do begin
      Cost:= FindX(X, Y, Z);
      if Cost < BestCost then begin
        BestCost:= Cost;
        aX:= X; aY:= Y; aZ:= Z;
      end;
      X:= NodeNext;
    end;
    Result:= BestCost;
  end;
  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var NodeX, NodeY, NodeZ : TNode;
    Cost, CostX : integer;

begin  { GenHamilton_Quick }

  // Для орграфа используем основной метод:
  if mDirect then begin
    Result:= GenHamilton(aCost);
    Exit;
  end;

  // Находим вершину X и двух её ближайших соседей Y, Z таких,
  // что сумма расстояний от X к двум этим соседям Y и Z минимальна

  CostX:= FindNode(NodeX, NodeY, NodeZ);

  // Строим кратчайшую цепь от вершины Y к вершине Z
  // с исключением вершины X

  Result:= GenHamPathStrip(NodeY, NodeZ, NodeX, Cost);
  if Cost < 0 then Exit;

  aCost:= Cost + CostX;  // общая стоимость контура

  // Вставляем вершину в разрез между Z и Y, замыкая тем самым контур
  Result.Push(NodeX);

  // Вращаем буфер, пока первая вершина графа не сместится в его начало
  NodeX:= NodeFirst;
  repeat
    NodeY:= Result.Get as TNode;
    if NodeY <> NodeX then Result.Put(NodeY);
  until NodeY = NodeX;
  Result.Push(NodeY);
end;
```

Хотите проверить идею на практике? К вашим услугам программа для тестирования метода (листинг 33-4). Здесь в качестве идеального решения для небольших графов принят результат полного перебора, а для крупных — то из двух решений, что даёт меньшую цену. Тестируются полные неориентированные (симметричные) графы, длина рёбер которых равномерно распределена на интервале от 1 до 99.

Листинг 33-4 — Программа для сравнения основного и ускоренного алгоритмов поиска гамильтонова контура в неориентированных графах

```
{$APPTYPE CONSOLE}
uses
  SysUtils,  DateUtils,
  Assembly in '..\Common\Assembly.pas',
  Graph in '..\Common\Graph.pas',
  GrChars in '..\Common\GrChars.pas',
  Items in '..\Common\Items.pas',
  Root in '..\Common\Root.pas',
  SetList in '..\Common\SetList.pas',
  SetUtils in '..\Common\SetUtils.pas';

procedure Test(aRetry, aNodes: integer);

// Размер массивов для накопления погрешностей
const CArrSize = 50; // 50 квантов по 0,5% (всего 25 %)

var Gr : TGraph;           // исследуемый граф
    Res : TBuffer;         // результат - гамильтонов контур
    Cost, CostF: integer; // стоимости

    TF, TH, TQ : integer;    // время накопленное
    MaxH, MaxQ : extended;   // максимальные погрешности
    SumH, SumQ : extended;   // суммы погрешностей
    Start : TDateTime;       // для засечки времени

    Cnt : integer;       // счётчик решений
    Delta : integer;     // абсолютное отклонение
    pc :  extended;      // отношение в %
    Index: integer;      // индекс в статистике

    // массивы для накопления распределения погрешностей (25% с шагом 0,5%)
    ArrH, ArrQ : array [0..CArrSize] of integer;

   // Вывод статистики
  procedure Expo(const aFileName: String);
  const CLine = '-------------------------------------------------------';
  var i : integer;
      exist: boolean;
  begin
    exist:= true;
    if aFileName <> '' then begin
      Assign(Output, aFileName);
      exist:= FileExists(aFileName);
      if exist then Append(Output) else Rewrite(Output)
    end;
    if not exist then Writeln(aFileName);
    Writeln(CLine);
    Writeln('Retry=', aRetry:4,  '  Nodes=', aNodes:4);
    Writeln(CLine);
    Writeln('                  F         H         Q');
    // Среднее время, мс
    Writeln('Time (ms)= ', TF/aRetry:10:1, TH/aRetry:10:1, TQ/aRetry:10:1);
    // максимальное отклонение
    Writeln('Max  (%%)= ', 0.0:10:1, (MaxH-100):10:1, (MaxQ-100):10:1);
    // средне квадратичное отклонение
    Writeln('Aver (%%)= ', 0.0:10:1,
             Sqrt(SumH/aRetry):10:1,
             Sqrt(SumQ/aRetry):10:1);
```

```
    Writeln(CLine);
    // Таблицы распределения
    if Abs(aNodes) < 15 then begin
      for i:= 0 to CArrSize-1 do Writeln(i:2, #9, 100*ArrH[i]/aRetry:6:1);
      Writeln(CLine);
    end;
    if not Gr.mDirect then begin
      for i:= 0 to CArrSize-1 do Writeln(i:2, #9, 100*ArrQ[i]/aRetry:6:1);
      Writeln(CLine);
    end;
    if aFileName <> '' then begin
      Close(Output);  Assign(Output, '');    Rewrite(Output);
    end;
  end;

begin  { Test }
    FillChar(ArrH, SizeOf(ArrH), 0);
    FillChar(ArrQ, SizeOf(ArrH), 0);
    // время накопленное
    TF:=0; TH:=0; TQ:=0;
    // максимальные погрешности
    MaxH:= 0; MaxQ:= 0;
    // суммы погрешностей
    SumH:= 0; SumQ:= 0;
    for Cnt:= 1 to aRetry do begin
      Gr:= TGraphChars.GenFull((aNodes>0), 0, 99, Abs(aNodes));
      {-------- Full -------}
      CostF:= 0;
      if Abs(aNodes) < 15 then begin
        Start:= Now;
        Res:= Gr.GenHamilton_Full(CostF);
        Inc(TF, MilliSecondsBetween(Start, Now));
        Res.Free;
      end;
      {-------- Main -------}
      Start:= Now;
      Res:= Gr.GenHamilton(Cost);
      Inc(TH, MilliSecondsBetween(Start, Now));
      Res.Free;
      if CostF=0 then CostF:= Cost; // если не вычислялся перебор
      Delta:= Cost - CostF;         // абсолютное отклонение
      if Delta=0 then Write('.') else Write('*');
      pc:= 100 * Cost / CostF;      // относительное отклонени pc >= 100 %
      if MaxH < pc then MaxH:= pc;  // максимальное отклонение
      SumH:= SumH + (pc-100)*(pc-100);
      // Формируем таблицу распределения с шагом 0,5 %:
      Index:= Round(2*(pc-100));
      if Index < CArrSize then Inc(ArrH[Index]);
      {-------- Quick -------}
      if not Gr.mDirect then begin
        Start:= Now;
        Res:= Gr.GenHamilton_Quick(Cost);
        Inc(TQ, MilliSecondsBetween(Start, Now));
        Res.Free;
        Delta:= Cost - CostF;    // абсолютное отклонение
        // Из двух решений (Cost и CostF) выбираем лучшее
        if Delta<0 then begin
          Delta:= 0;
          CostF:= Cost;
        end;
        if Delta=0 then Write('.') else Write('@');
        pc:= 100 * Cost / CostF;      // относительное отклонени pc >= 100 %
```

```
        if MaxQ < pc then MaxQ:= pc;  // максимальное отклонение
        SumQ:= SumQ + (pc-100)*(pc-100);
        // Формируем таблицу распределения с шагом 0,5 %:
        Index:= Round(2*(pc-100));
        if Index < CArrSize then Inc(ArrQ[Index]);
      end;
      Gr.Free;
    end; // for
    Writeln(#7);
    // Вывод статистики
    Expo('');   Expo('Out.txt');
end;
{-------------------------------------------------------}
var Nodes: integer;
    Retry : integer;
begin
    Write('Retry= '); Readln(Retry);
    if Retry = 0 then Exit;
    repeat
      Write('Nodes= '); Readln(Nodes);
      if Abs(Nodes) < 4 then Break;
      Test(Retry, Nodes);
    until false;
end.
```

Теперь рассмотрим результаты испытаний и вначале оценим точность основного и ускоренного методов в сравнении с идеалом, полученным полным перебором. В табл. 33-2 дан усреднённый результат, полученный для основного метода на небольших графах с количеством вершин от 11 до 14. Всего здесь испытано 4 тысячи полных симметричных графов (по тысяче для каждого размера), длина рёбер у графов распределена равномерно в интервале от 1 до 99.

Табл. 33-2 — Отклонение стоимости от идеала для *ОСНОВНОГО* алгоритма (на малых графах)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	95,75	5,0	0,23
0,5	0,88	5,5	0,08
1,0	0,63	6,0	0,05
1,5	0,55	6,5	0,05
2,0	0,30	7,0	0,05
2,5	0,35	7,5	0,05
3,0	0,38	8,0	0,08
3,5	0,25	8,5	0,05
4,0	0,10	9,0	0,00
4,5	0,13	9,5	0,03

Результаты, полученные для ускоренного метода, показаны в табл. 33-3.

Табл. 33-3 — Отклонение стоимости от идеала для *ускоренного* алгоритма (на малых графах)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	41,00	5,0	2,45	10,0	1,43
0,5	3,13	5,5	2,65	10,5	1,50
1,0	2,15	6,0	2,08	11,0	1,65
1,5	2,75	6,5	1,93	11,5	1,03
2,0	2,68	7,0	1,78	12,0	1,00
2,5	2,23	7,5	1,40	12,5	0,90
3,0	2,58	8,0	1,58	13,0	1,00
3,5	2,58	8,5	1,65	13,5	0,85
4,0	2,15	9,0	1,13	14,0	0,83
4,5	1,88	9,5	1,05	14,5	0,85

На рис. 33-3 и рис. 33-4 всё это представлено в графической форме. Первый столбец, соответствующий точному совпадению, на диаграммах не показан.

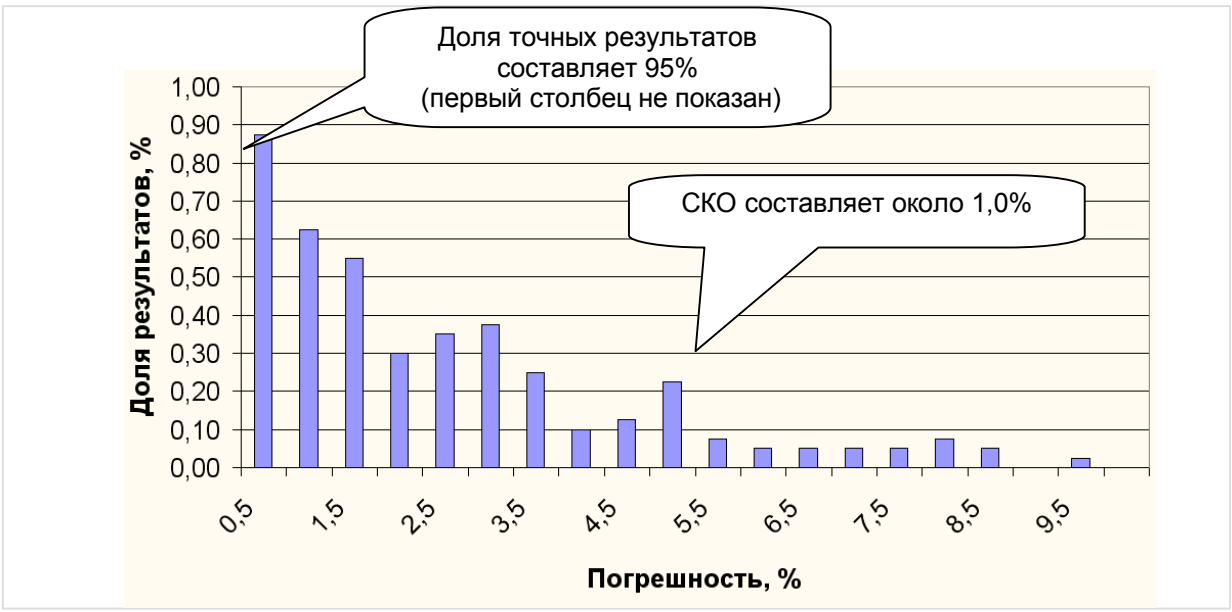


Рис. 33-3 — Распределение погрешностей для *ОСНОВНОГО* алгоритма (количество вершин от 11 до 14)

Оценим точность алгоритмов по двум параметрам: а) по доле точных совпадений стоимости с идеалом, и б) по среднеквадратичному отклонению стоимости от идеала. Доля точных совпадений для основного и ускоренного методов составила соответственно 95% и 41%, а среднеквадратичные отклонения — 1% и 8,5%, что говорит о существенно сниженной точности ускоренного алгоритма (за всё приходится платить!). Однако ж в сравнении с жадным алгоритмом эта точность весьма высока.

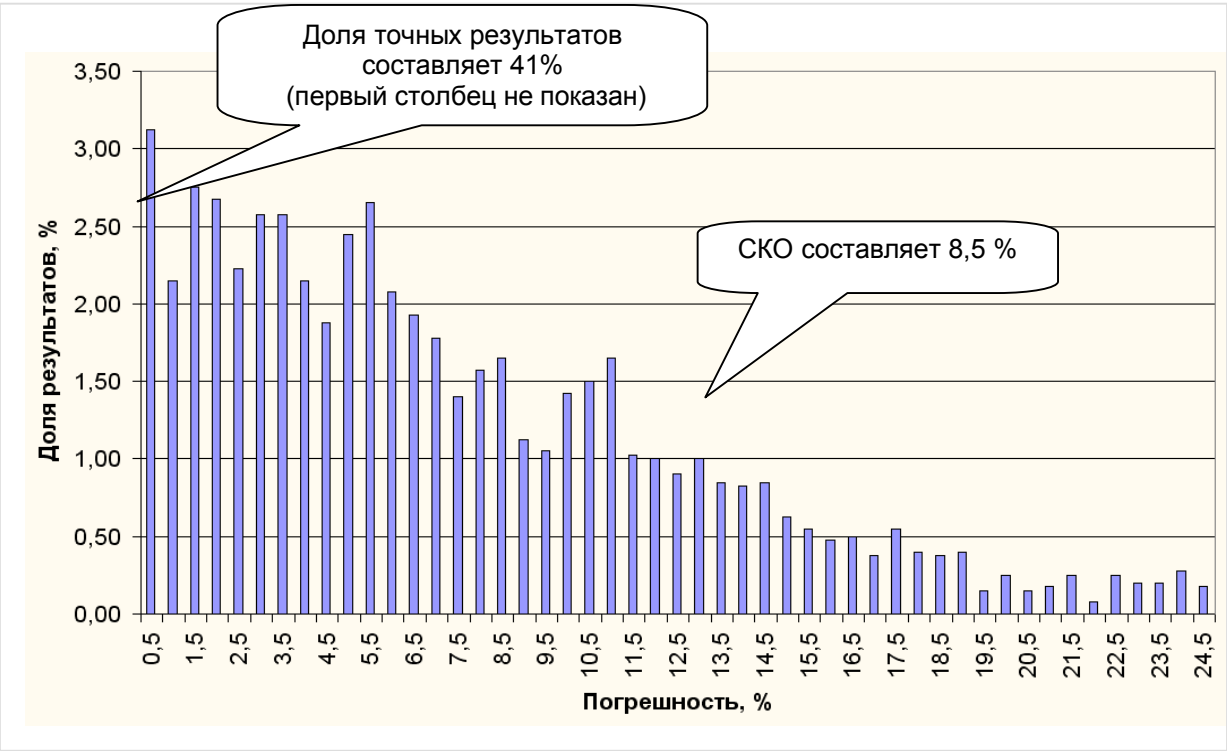


Рис. 33-4 — Распределение погрешностей для *ускоренного* алгоритма (количество вершин от 11 до 14)

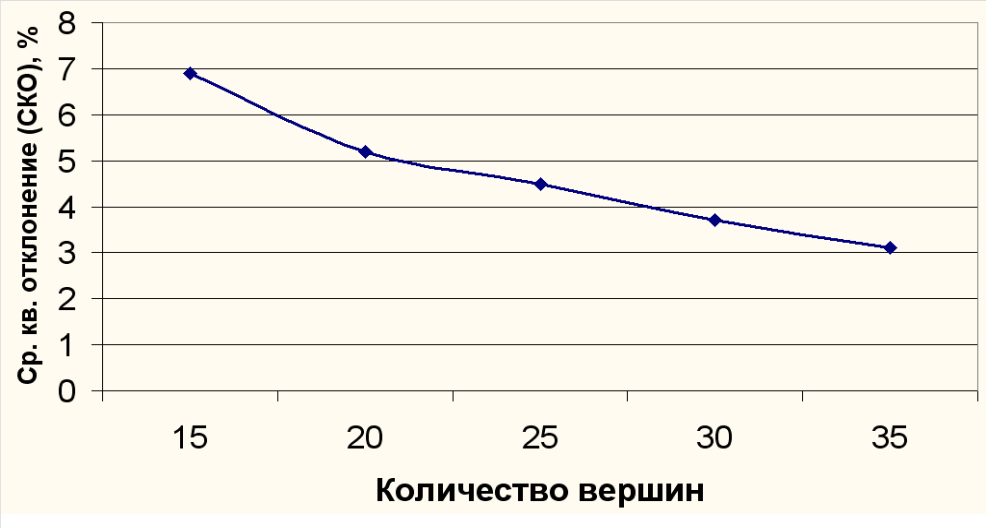
А какова точность при большем количестве вершин? Увы, здесь не доступен идеал, даваемый полным перебором, и потому при сравнении основного и ускоренного методов в качестве условного идеала возьмём результат с меньшей стоимостью (из этих двух). Ниже даны усреднённые результаты обработки графов с числом вершин 15, 20, 25, 30 и 35; обработано по тысяче симметричных случайных полных графа каждого размера, длина их рёбер распределена равномерно в интервале от 1 до 99.

Табл. 33-4 — Отклонение от условного идеала ускоренного алгоритма (на графах от 15 до 35 вершин)

Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %	Отклонение от идеала, %	Доля результатов, %
0,0	39,36	5,0	2,68	10,0	0,70
0,5	5,90	5,5	2,10	10,5	0,80
1,0	5,16	6,0	1,84	11,0	0,78
1,5	5,38	6,5	1,96	11,5	0,52
2,0	4,28	7,0	1,00	12,0	0,38
2,5	4,66	7,5	1,44	12,5	0,50
3,0	3,96	8,0	1,32	13,0	0,42
3,5	3,10	8,5	1,16	13,5	0,14
4,0	2,98	9,0	0,76	14,0	0,22
4,5	3,46	9,5	0,74	14,5	0,38

Рис. 33-5 — Распределение погрешностей для ускоренного алгоритма при количестве вершин от 15 до 35 (в сравнении с основным алгоритмом)

Здесь отметим, что с ростом числа вершин точность ускоренного метода, по крайней мере, не ухудшается, а среднеквадратичное отклонение даже уменьшается (рис. 33-6).



N	СКО, %
15	6,9
20	5,2
25	4,5
30	3,7
35	3,1

Рис. 33-6 — Зависимость среднеквадратичного отклонения от количества вершин для ускоренного метода

Итак, мы выяснили, что точность ускоренного метода в сравнении с основным, как и ожидалось, оказалась заметно ниже. Оправдано ли это снижение точности повышением скорости? В табл. 33-5 и на рис. 33-7 дано усреднённое время обработки выше упомянутых графов.

Табл. 33-5 — Среднее время обработки полных симметричных графов разных размеров, ms

Количество вершин	Среднее время поиска, ms		Отношение времён
	Основной	Ускоренный	
15	**16,7**	**7,6**	**2,2**
20	**67,4**	**29,4**	**2,3**
25	**196,6**	**76,8**	**2,6**
30	**602,0**	**161,0**	**3,7**
35	**1454,0**	**303,0**	**4,8**

Таким образом, ускоренный алгоритм отвечает своему названию: при поиске гамильтоновых контуров в *симметричных* графах он даёт ощутимый выигрыш во времени, который только растёт с ростом размера графа. Так, если на **15** вершинах наблюдается двукратный выигрыш, то при **35** — уже пятикратный.

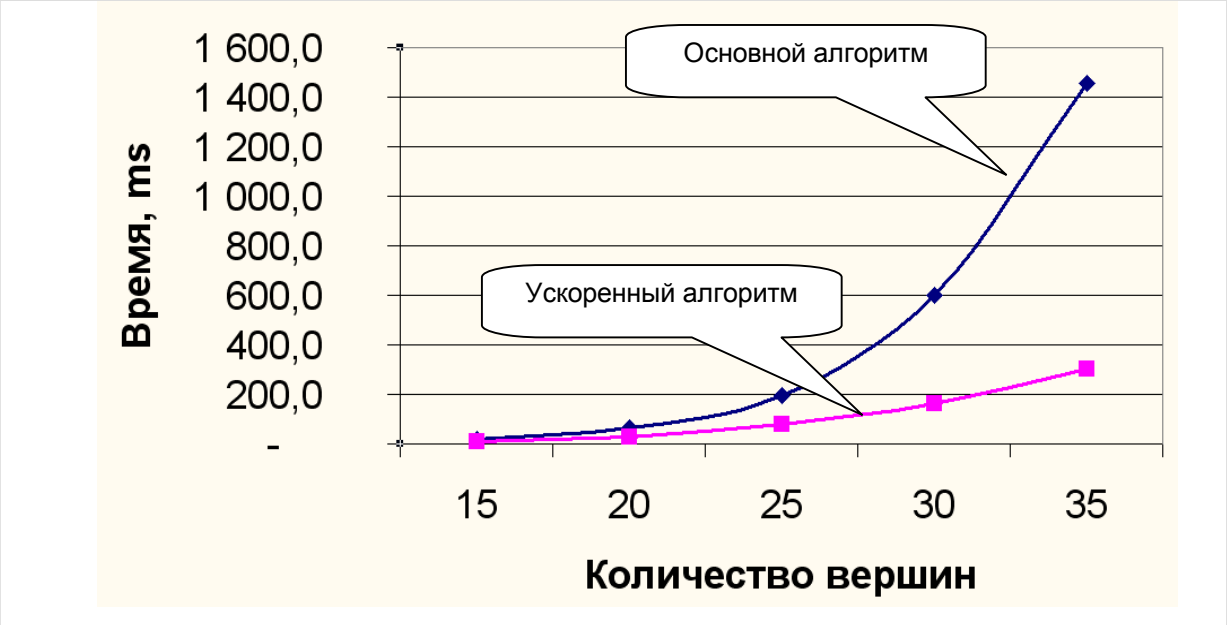


Рис. 33-7 — Среднее время обработки симметричных полных графов разных размеров

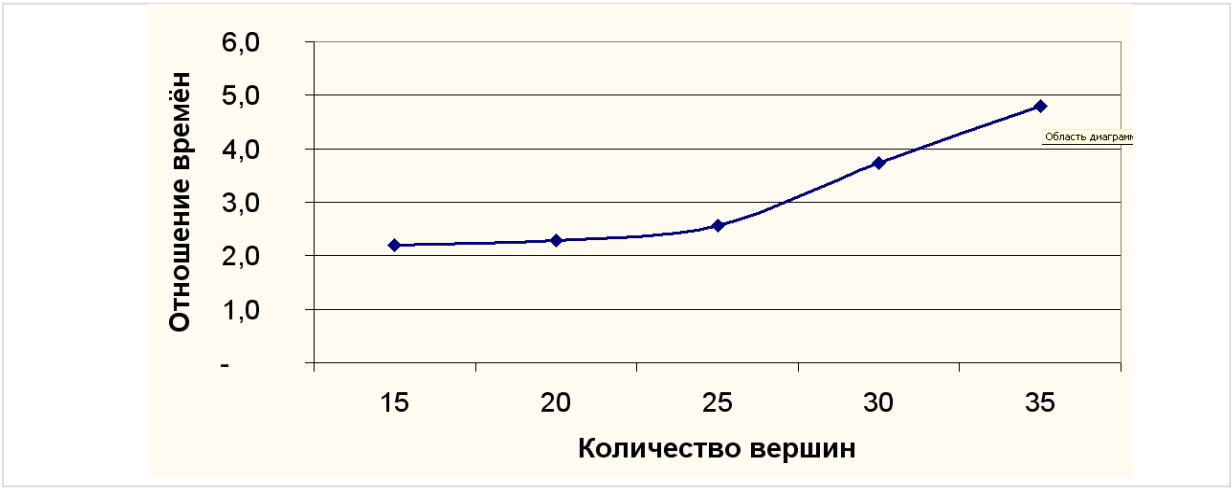


Рис. 33-8 — Отношение времени обработки симметричных графов основным и ускоренным алгоритмами

33.3. Итоги

33.3.1. Для разомкнутой и замкнутой задачи Гамильтона применимы два принципиально разных алгоритма: алгоритм штрафования вершин (для неориентированных графов) и алгоритм поиска замкнутых контуров (на основе венгерского метода). Однако в чистом виде эти основные алгоритмы не решают всего круга гамильтоновых задач.

33.3.2. *Разомкнутую* задачу для *орграфа* можно решить слегка дополненным методом, разработанным для замкнутой задачи. Модификация состоит в добавлении к исходному графу одной вспомогательной вершины с двумя дугами нулевой длины.

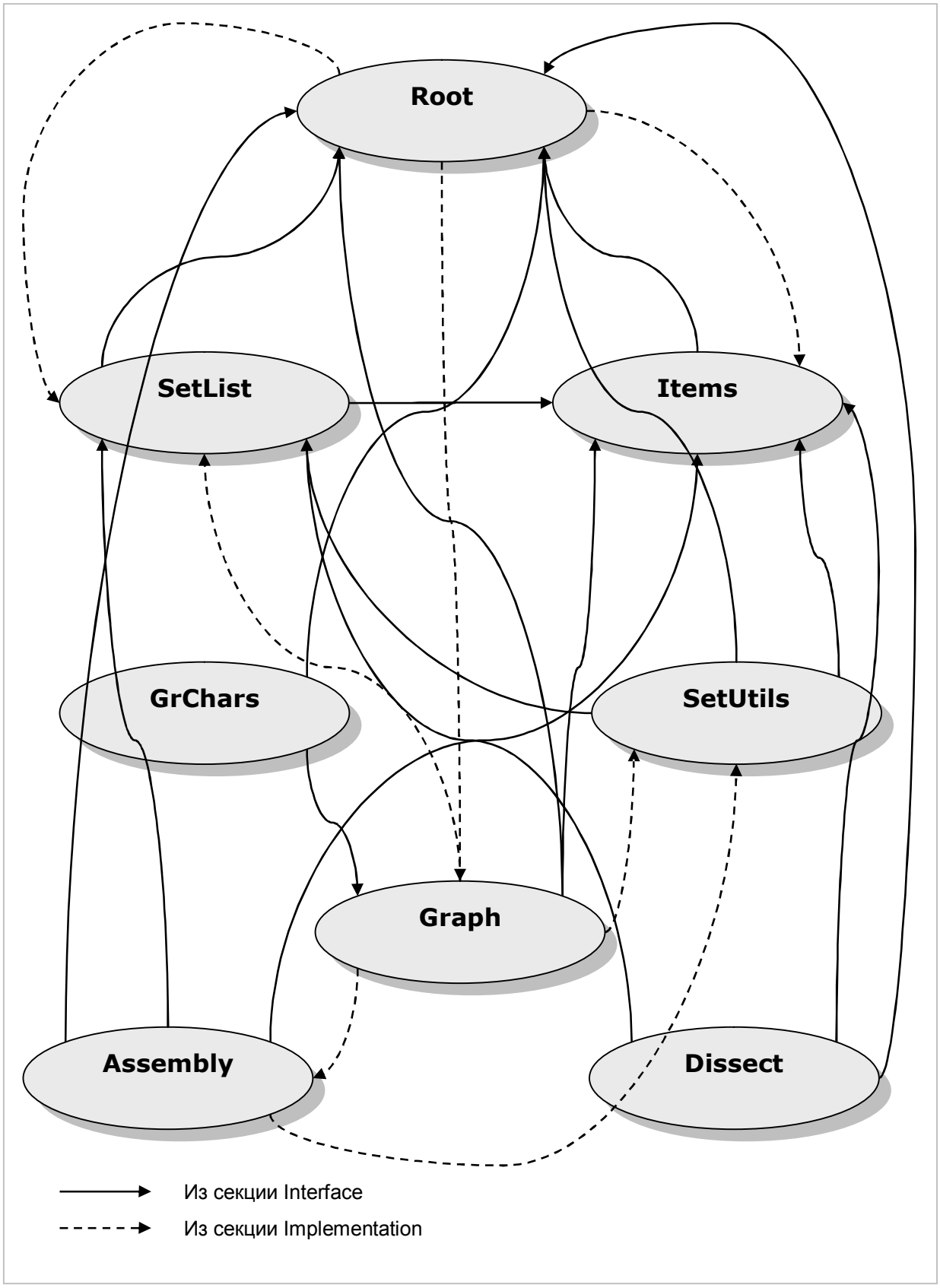
33.3.3. *Замкнутая* задача для *неориентированного графа* быстрее решается методом, разработанным для разомкнутой задачи. Для этого выбирают подходящую вершину и временно отключают её, превращая контур в гамильтонов путь. При этом несколько снижается точность решения.

33.4. Задания

33.5. Что почитать

№	Автор(ы)	Название	Главы, страницы
1	Андреева Е.В. Босова Л.Л. Фалина И.Н.	Математические основы информатики	
2	Басакер Р. Саати Т.	Конечные графы и сети	
3	Вирт Н.	Алгоритмы + Структуры данных = Программы	
4	Дарахлевидзе П., Марков Е.	Программирование в Delphi 7	
5	Деревенец О.В.	Песни о Паскале	
6	Кормен Т., Лейзерсон Ч., Ривест Р.	Алгоритмы. Построение и анализ	
√ 7	Кристофидес Н.	Теория графов. Алгоритмический подход	Стр. 273
8	Липский В.	Комбинаторика для программистов	
9	Майника Э.	Алгоритмы оптимизации на сетях и графах	
10	Макконнелл Дж.	Основы современных алгоритмов	
11	Турчин В.Ф.	Феномен Науки	
12	Хаггарти Р.	Дискретная математика для программистов	

Приложение A
Взаимные ссылки модулей



Приложение B
Модуль Root

```
unit Root;
{$I Common.inc}
//*************************************************************************
//                          Корневой модуль                                *
//                                                                         *
// Содержит базовые объекты:                                               *
// TItem -- элемент                                                        *
// TSet  -- абстрактное множество                                          *
// TCostSet -- множество с оценкой для решения задач ЗНП и ЗНР             *
//*************************************************************************

interface

type

  // Перечисление результатов сравнения объектов

  TCompare = (cmpEq,     // совпадают или равны
              cmpLess,   // меньше
              cmpGreate, // больше
              cmpIncomp  // несравнимы (Incomparable)
             );

  // Базовый класс "элемент" -- прародитель всех классов

  TItem = class (TObject)
    public
      function Compare(arg: TItem): TCompare; virtual;
      function Copy: TItem;  virtual;{ abstract;}
      procedure Print(var aFile: TextFile); virtual; abstract;
      procedure Expo;
  end;

  // Класс "абстрактное множество"

  TSet = class (TItem)
    protected
      mCount: Longint;  // количество элементов множества
      procedure Clr(aDestroy: boolean); virtual; abstract;
    public
      // Базовые операции с множествами:
      procedure Add(arg : TSet); virtual; abstract;
      procedure Sub(arg : TSet); virtual; abstract;
      procedure Mul(arg : TSet); virtual; abstract;
      procedure ExOr(arg : TSet); virtual; abstract;
      function  TestIntersect(arg: TSet): boolean;
      function  Compare(arg: TItem): TCompare; override;
      function  Exist(arg : TItem): boolean; virtual; abstract;
      // Очистка
      procedure Clear;          // очистка без уничтожения элементов
      procedure ClrAndDestroy;  // очистка с уничтожением элементов
      // Вставка, удаление, копирование элементоа
      function  Insert(arg: TItem): boolean; virtual; abstract;
      procedure Delete(arg: TItem); virtual; abstract;
      procedure CopyItems(arg: TSet);
      // Последовательный перебор элементов
```

```
      function  GetFirst: TItem; virtual; abstract;
      function  GetNext: TItem; virtual; abstract;
      // Выбор по индексу и по элементу
      function  GetItem(index: integer): TItem;
      function  GetObject(aItem: TItem): TItem;
      // Сохранение-восстановление позиции перебора
      procedure PositionPush;  virtual; abstract;
      procedure PositionPop;  virtual; abstract;
      // Прочие операции
      procedure CoverToDissect; // Преобразование покрытия в разбиение
      function  GetCount: integer;
      function  Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      destructor Destroy;  override;
  end;

  // Класс "множество с оценкой", используется при поиске
  // наименьших разбиений (ЗНР) и покрытий (ЗНП)

  TCostSet = class (TItem)
    protected
      procedure Clr(aDestroy: boolean);
    public
      mCost: integer;       // оценка, стоимость элемента
      mSet : TSet;          // ссылка на множество
      mFlag  : boolean;     // флажок для пометок
      mDestroy : boolean;   // определяет способ уничтожения объекта
      constructor Create(aCost: integer; aSet: TSet; aDestroy : boolean);
      constructor CreateEmpty;
      destructor  Destroy;  override;
      function    Copy: TItem; override;
      procedure   Clear;
      procedure   ClrAndDestroy;
      function    Compare(arg: TItem): TCompare; override;
      procedure   Print(var aFile: TextFile); override;
      procedure   Append(arg : TCostSet);
      procedure   Insert(arg : TCostSet);
  end;

// Функция для создания множества

function CreateSet: TSet;

////////////////////////////////////////////////////////////////////////////
implementation
uses Items, SetList, Graph;
////////////////////////////////////////////////////////////////////////////

// Функция для создания множества

function CreateSet: TSet;
begin
  Result:= TSetList.Create;
end;

////////////////////////////////////////////////////////////////////////////
// TItem   -- Базовый класс "элемент" -- прародитель всех классов
////////////////////////////////////////////////////////////////////////////

function TItem.Compare(arg: TItem): TCompare;
begin
  if arg = Self
```

```
    then Result:= cmpEq
    else Result:= cmpIncomp;
end;


function TItem.Copy: TItem;
begin
  Result:= nil
end;

procedure TItem.Expo;       // Метод отображения на экране
begin
  if Assigned(Self) then Print(Output);
end;
////////////////////////////////////////////////////////////////////////////
// TSet  -- базовый класс "множество"
////////////////////////////////////////////////////////////////////////////

// Сравнение двух множеств

function TSet.Compare(arg: TItem): TCompare;
var p, q : TItem;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // один и тот же объект

  // Если аргумент не множество, то сравнивать нельзя
  if not (arg is TSet) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;

  // Сравнение элементов множеств

  PositionPush;             // сохр. позицию в данном множестве
  (arg as TSet).PositionPush;  // сохр. позицию в аргументе
  p:= GetFirst;             // взять первый элемент в данном множестве
  q:= (arg as TSet).GetFirst;  // взять первый элемент в аргументе
  while Assigned(p) and Assigned(q) do begin  // пока можно сравнивать
    Result:= p.Compare(q);                    // сравнение элементов
    if Result <> cmpEq then Break;            // выход, если не одинаковы
    p:= GetNext;  q:=(arg as TSet).GetNext;   // следующая пара
  end;
  if Result=cmpEq then begin                // если последние эл. совпали
    if GetCount < (arg as TSet).GetCount
      then Result:= cmpLess                 // данный меньше аргумента
    else if GetCount > TSet(arg).GetCount
      then Result:= cmpGreate               // данный больше аргумента
  end;
  (arg as TSet).PositionPop;    // восст. позицию в аргументе
  PositionPop;              // восст. позицию в данном множестве
end;

// Копирование множества целиком

function TSet.Copy: TItem;
var p, q : TItem;
begin
  Result:= CreateSet;
  PositionPush;
  p:= GetFirst;
  while Assigned(p) do begin
```

```
    if p is TSet
      then q:= p.Copy          // вложенное множество
      else q:= p;              // элементарный объект
    (Result as TSet).Insert(q);
    p:= GetNext;
  end;
  PositionPop;
end;

// Копирование элементов из аргумента в данное множество

procedure TSet.CopyItems(arg: TSet);
var t: TItem;
begin
  Clear;     // очистка множества без уничтожения элементов
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    Self.Insert(t);
    t:= arg.GetNext;
  end;
  arg.PositionPop;
end;

// очистка без уничтожения элементов

procedure TSet.Clear;
begin
  Clr(false)
end;

// очистка с уничтожением элементов

procedure TSet.ClrAndDestroy;
begin
  Clr(true)
end;

// Деструктор

destructor TSet.Destroy;
begin
  Clear;   // очистка без уничтожения элементов
  inherited;
end;

// Преобразование покрытия в разбиение.
// Элементами множества должны быть TCostSet!

procedure TSet.CoverToDissect;
var   i, j : integer;  // индексы
    Si, Sj : TCostSet; // текущие элементы
begin
  for i:= 1 to GetCount-1 do begin
    Si:= GetItem(i) as TCostSet;
    // Вычитаем из текущего подмножества все последующие
    for j:= i+1 to GetCount do begin
      Sj:= GetItem(j) as TCostSet;
      Si.mSet.Sub(Sj.mSet);
    end;
  end;
end;
```

```
// Вывод множества в текстовый файл.
// Вспомогательные переменные PrintLevel и OldFlag
// служит для формирования отступов
// при печати вложенных множеств

var PrintLevel : integer = 0;
    OldFlag: boolean;

procedure TSet.Print(var aFile: TextFile);
var t : TItem;      // очередной элемент
    S : string;     // отступ из пробелов
    i : integer;
    flag: boolean;
begin
  S:=''; for i:=1 to 2*PrintLevel do S:= S+#32;
  Write(aFile,S+'{');
  if PrintLevel=0 then OldFlag:= true;
  Inc(PrintLevel);
  flag:= false;
  PositionPush;
  t:= GetFirst;
  while Assigned(t) do begin
    flag:= (t is TSet) or
           (t is TCostSet) or
           (t is TGraph) or
           (t is TNode);
    if flag and OldFlag then Writeln(aFile);
    t.Print(aFile);
    t:= GetNext;
  end;
  PositionPop;
  if flag
    then Writeln(aFile,S+'} : ', mCount)
    else Writeln(aFile,' } : ', mCount);
  OldFlag:= flag;
  Dec(PrintLevel);
end;

// Проверка пересечения двух множеств

function TSet.TestIntersect(arg: TSet): boolean;
var  Temp: TSet;   // для проверки пересечения
begin
  Temp:= CreateSet;
  Temp.CopyItems(Self);
  Temp.Mul(arg);
  Result:= Temp.GetCount > 0;
  Temp.Free;
end;

function TSet.GetCount: integer;
begin
  Result:= mCount;
end;

// Доступ к элементу множества по его индексу
// index = 1..GetCount

function TSet.GetItem(index: integer): TItem;
begin
  Result:= nil;
```

```
  if (index<1) or (index>GetCount) then exit;
  PositionPush;              // сохр. позицию
  Result:= GetFirst;         // взять первый
  while index>1 do begin     // переход по цепочке
    Dec(Index);
    Result:= GetNext;
  end;
  PositionPop;               // восст. позицию
end;

// Доступ к элементу множества по содержимому

function TSet.GetObject(aItem: TItem): TItem;
begin
  PositionPush;              // сохр. позицию
  Result:= GetFirst;         // взять первый
  while Assigned(Result) and
        (Result.Compare(aItem) <> cmpEq)
    do Result:= GetNext;     // переход по цепочке
  PositionPop;               // восст. позицию
end;

////////////////////////////////////////////////////////////////////////////
// TCostSet -- множество с оценкой
////////////////////////////////////////////////////////////////////////////

constructor TCostSet.Create(aCost: integer; aSet: TSet; aDestroy : boolean);
begin
  inherited Create;
  mCost:= aCost;      // оценка
  mSet := aSet;       // ссылка на множество
  if not Assigned(mSet) then begin
    mDestroy:= true;
    mSet:= CreateSet;
  end;
  if aDestroy then mDestroy:= true;  // множество уничтожается с объектом
end;

constructor TCostSet.CreateEmpty;
begin
  inherited Create;
  mDestroy:= true;
  mSet:= CreateSet;
end;

function TCostSet.Copy: TItem;
begin
  Result:= TCostSet.Create(mCost, mSet.Copy as TSet, mDestroy);
  (Result as TCostSet).mDestroy:= true;
end;

destructor TCostSet.Destroy;
begin
  if mDestroy then mSet.Free;
  inherited;
end;


procedure TCostSet.Clr(aDestroy: boolean);
begin
  if aDestroy
    then mSet.ClrAndDestroy
```

```
    else mSet.Clear;
  mCost:=0;
end;

procedure TCostSet.Clear;
begin
  Clr(false)
end;

procedure TCostSet.ClrAndDestroy;
begin
  Clr(true)
end;

// Добавление подмножеств

procedure TCostSet.Append(arg: TCostSet);
var CS : TCostSet;
     i : integer;
begin
  if not Assigned(arg) then Exit;
  for i:= 1 to arg.mSet.GetCount do begin
    CS:= TCostSet(arg.mSet.GetItem(i));
    Insert(CS);
  end;
end;

// Вставка нового элемента (подмножества) с корректировкой цены

procedure TCostSet.Insert(arg: TCostSet);
begin
  if not Assigned(arg) then Exit;
  Inc(mCost, arg.mCost);
  mSet.Insert(arg);
end;

// Сравнение множеств с оценкой:

function TCostSet.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin
    Result:= cmpEq;
    Exit;
  end;
  Result:= cmpIncomp;  // начальное значение
  if not Assigned(arg) then Exit;
  if not (arg is TCostSet) then Exit;
  // Сравниваем цены:
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене
    // сравнением мощности множеств
    if mSet.GetCount > (arg as TCostSet).mSet.GetCount
      then Result:= cmpLess
    else if mSet.GetCount < (arg as TCostSet).mSet.GetCount
      then Result:= cmpGreate
      // Если мощности одинаковы сравниваем сами множества
      else Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали,
    // сравнениваем по удельной цене (дешёвые - в начало)
    if mCost * (arg as TCostSet).mSet.GetCount <
      (arg as TCostSet).mCost * mSet.GetCount
```

```
      then  Result:= cmpLess
      else
        if mCost * (arg as TCostSet).mSet.GetCount >
            (arg as TCostSet).mCost * mSet.GetCount
          then   Result:= cmpGreate
          else  { сравнение мощностей элементов-множеств }
            if mSet.GetCount > (arg as TCostSet).mSet.GetCount
              then Result:= cmpLess
              else
                if mSet.GetCount < (arg as TCostSet).mSet.GetCount
                  then Result:= cmpGreate
  end;
end;


procedure TCostSet.Print(var aFile: TextFile);
var t: TItem;
begin
  t:= mSet.GetItem(1);
  if Assigned(t) and (t is ClassType) then begin
    Writeln(aFile, 'Cost= ', mCost);  // печатаем оценку и новую строку
  end else begin
    Write(aFile, ' Cost=', mCost:4, '  ');  // печатаем только оценку
  end;
  mSet.Print(aFile)
end;

end.
```

Приложение C
Модуль Items

```
unit Items;
{$I Common.inc}
//*************************************************************************
//                        Модуль с элементами                             *
//                                                                        *
// Содержит объекты:                                                      *
// TItemChar -- элемент-символ                                            *
// TItemStr  -- элемент-строка                                            *
// TItemNum  -- элемент-число                                             *
// TBuffer   -- универсальный буфер                                       *
//*************************************************************************

interface

uses Root;

type

  // TItemChar -- элемент-символ
  TItemChar = class (TItem)
    private
      mData: Char;
    public
      constructor Create(arg: Char);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: Char;
  end;

  // TItemStr  -- элемент-строка
  TItemStr = class(TItem)
    private
      mData: String;
    public
      constructor Create(arg: string);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: string;
  end;

  // TItemNum  -- элемент-число
  TItemNum = class (TItem)
    private
      mData: integer;
    public
      constructor Create(arg: integer);
      function Compare(arg: TItem): TCompare; override;
      function Copy: TItem; override;
      procedure Print(var aFile: TextFile); override;
      function GetData: integer;
  end;

  // Вспомогательная структура для организации
  // односвязного списка
```

```
  PStackRec = ^TStackRec;
  TStackRec = record
    mItem : TItem;             // элемент списка
    mNext : PStackRec;         // ссылка на следующий
  end;

  // Класс для организации буферов: стеков и очередей

  TBuffer = class (TItem)
    private
      mCount: integer;                // счётчик элементов
      mHead, mQue: PStackRec;         // указатели на начало и конец буфера
      procedure Clr(aDestroy: boolean);
    public
      destructor Destroy;  override;
      function Copy: TItem; override; // создание копии
      procedure Put(arg: TItem);      // занесение в очередь
      function Get: TItem;            // извлечение из очереди
      procedure Push(arg: TItem);     // занесение в стек
      function Pop: TItem;            // извлечение из стека
      procedure Reversion;            // реверс буфера
      function GetByIndex(aIndex: integer): TItem; // доступ по индексу
      function GetCount: integer;     // количество элементов в буфере
      procedure Clear;                // очистка без уничтожения объектов
      procedure ClrAndDestroy;        // очистка с уничтожением объектов
      procedure Print(var aFile: TextFile); override; // вывод в файл
  end;

////////////////////////////////////////////////////////////////////////////////
implementation
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
// TItemChar -- элемент-символ
////////////////////////////////////////////////////////////////////////////////

function TItemChar.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemChar) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemChar).mData = mData
    then Result:=cmpEq
    else if (arg as TItemChar).mData > mData
            then Result:= cmpLess
            else Result:= cmpGreate;
end;

constructor TItemChar.Create(arg: Char);
begin
  inherited Create;
  mData:= arg;
end;

function TItemChar.Copy: TItem;
begin
  Result:= TItemChar.Create(mData);
end;
```

```
procedure TItemChar.Print(var aFile: TextFile);
begin
  Write(aFile, mData:2)
end;

function TItemChar.GetData: Char;
begin
  Result:= mData
end;

////////////////////////////////////////////////////////////////////////////
// TItemStr  -- элемент-строка
////////////////////////////////////////////////////////////////////////////

function TItemStr.Copy: TItem;
begin
  Result:= TItemStr.Create(mData);
end;

constructor TItemStr.Create(arg: string);
begin
  inherited Create;
  mData:= arg
end;

function TItemStr.GetData: string;
begin
  Result:= mData;
end;

function TItemStr.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemStr) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
  if (arg as TItemStr).mData <> mData
    then if (arg as TItemStr).mData > mData
           then Result:= cmpLess
           else Result:= cmpGreate
end;

procedure TItemStr.Print(var aFile: TextFile);
begin
  Write{ln}(aFile, ' ', mData);
end;

////////////////////////////////////////////////////////////////////////////
// TItemNum  -- элемент-число
////////////////////////////////////////////////////////////////////////////

function TItemNum.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if Self=arg then Exit;  // совпадают
  if not (arg is TItemNum) then begin
    Result:=cmpIncomp;    // несравнимы
    Exit;
  end;
```

```
  if (arg as TItemNum).mData <> mData
    then if (arg as TItemNum).mData > mData
           then Result:= cmpLess
           else Result:= cmpGreate
end;


function TItemNum.Copy: TItem;
begin
  Result:= TItemNum.Create(mData);
end;

constructor TItemNum.Create(arg: integer);
begin
  inherited Create;
  mData:= arg
end;

function TItemNum.GetData: integer;
begin
  Result:= mData;
end;

procedure TItemNum.Print(var aFile: TextFile);
begin
  Write(aFile,' ', mData);
end;

////////////////////////////////////////////////////////////////////////////
// TBuffer -- класс для организации очередей и стеков
////////////////////////////////////////////////////////////////////////////

// Создание копии буфера из существующего

function TBuffer.Copy: TItem;
var t: TItem;
    i: integer;
begin
  Result:= TBuffer.Create;
  for i:= 1 to GetCount do begin
    t:= Get;
    Put(t);
    TBuffer(Result).Put(t);
  end;
end;

// Уничтожение буфера без уничтожения элементов

destructor TBuffer.Destroy;
begin
  Clear;
  inherited;
end;

// Очистка буфера

procedure TBuffer.Clr(aDestroy: boolean);
var i : integer;
    t : TItem;
begin
  for i:= 1 to mCount do begin
    t:= Get;
```

```
    if aDestroy then t.Free
  end;
end;

// Очистка с уничтожением элементов

procedure TBuffer.ClrAndDestroy;
begin
  Clr(true)
end;

// Очистка без уничтожения

procedure TBuffer.Clear;
begin
  Clr(false)
end;

// Получение счётчика элементов

function TBuffer.GetCount: integer;
begin
  Result:= mCount;
end;

// Push - помещение в стек (в начало буфера)

procedure TBuffer.Push(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= mHead;
  mHead:= r;
  Inc(mCount);
  // Обновляем указатель на конец списка
  if not Assigned(mQue) then mQue:= r;
end;

// Put - помещение в очередь (в конец буфера)

procedure TBuffer.Put(arg: TItem);
var r: PStackRec;
begin
  New(r);
  r^.mItem:= arg;
  r^.mNext:= nil;
  // Присоединяем к концу списка
  if Assigned(mQue) then mQue^.mNext:= r;
  mQue:= r;
  Inc(mCount);
  // Обновляем указатель на начало
  if not Assigned(mHead) then mHead:= r;
end;

// Извлечение из очереди и стека (из начала списка )

function TBuffer.Get: TItem;
var r: PStackRec;
begin
  Result:= nil;
  if not Assigned(mHead) then Exit;
```

```
  r:= mHead;
  Result:= mHead^.mItem;
  mHead:= mHead^.mNext;
  Dispose(r);
  Dec(mCount);
  // Обновляем указатель на хвост
  if mCount=0 then mQue:= nil;
end;

// Извлечение из стека (синоним Get)

function TBuffer.Pop: TItem;
begin
  Result:= Get;
end;

// Доступ к очереди по индексу

function TBuffer.GetByIndex(aIndex: integer): TItem;
var  r: PStackRec;
begin
  Result:= nil;
  if (aIndex<1) or (aIndex>mCount) then Exit;
  r:= mHead;
  while Assigned(r) do begin
    Dec(aIndex);
    if aIndex=0 then Break;
    r:= r^.mNext;
  end;
  if Assigned(r) then Result:= r^.mItem;
end;

// Реверс буфера (перестановка в обратном порядке)

procedure TBuffer.Reversion;
var B : TBuffer;
    t : TItem;
begin
  B:= TBuffer.Create;
  t:= Get;
  while Assigned(t) do begin   B.Push(t);  t:= Get;  end;
  t:= B.Get;
  while Assigned(t) do begin   Put(t);   t:= B.Get;  end;
  B.Free;
end;

// Печать буфера

procedure TBuffer.Print(var aFile: TextFile);
var r: PStackRec;
begin
  Writeln(aFile, '(');
  r:= mHead;
  while Assigned(r) do begin
    r^.mItem.Print(aFile);
    r:= r^.mNext;
  end;
  Writeln(aFile, ') BufCount= ', mCount);
end;
end.
```

Приложение D
Модуль SetList

```
unit SetList;
{$I Common.inc}
//*************************************************************************
//        Реализация множества на базе односвязного списка                *
//                                                                        *
// Содержит объекты:                                                      *
// TSetList -- Множество, реализованное списком                           *
//*************************************************************************

interface

uses Root, Items;

type

  // Вспомогательная структура для организации односвязного списка

  PListRec = ^TListRec;
  TListRec = record
    mItem : TItem;
    mNext : PListRec;
  end;

  // Множество, реализованное списком

  TSetList = class (TSet)
    protected
      mHead: PListRec; // голова списка
      procedure Clr(arg: boolean); override;
    private
      mCurrent : PListRec;  // указатель на текущий элемент
      mStack : TBuffer;   // стек для хранения позиций чтения mCurrent
      procedure Del(var p, q : PListRec);
      procedure Ins(p: PListRec; var q: PListRec; arg: TItem);
    public
      destructor Destroy;  override;
      function Exist(arg : TItem): boolean; override;
      function Insert(arg: TItem): boolean; override;
      procedure Delete(arg: TItem); override;
      function GetFirst: TItem; override;
      function GetNext: TItem; override;
      procedure PositionPush;  override;
      procedure PositionPop;   override;
      procedure Add(arg : TSet); override;
      procedure Sub(arg : TSet); override;
      procedure Mul(arg : TSet); override;
      procedure ExOr(arg : TSet); override;
  end;

//////////////////////////////////////////////////////////////////////////
implementation
//////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
// TSetList -- Множество, реализованное списком
////////////////////////////////////////////////////////////////////////////

// Извлечение первого элемента списка

function TSetList.GetFirst: TItem;
begin
  mCurrent:= mHead;
  if Assigned(mCurrent)
    then Result:= mCurrent^.mItem
    else Result:= nil;
end;

// Извлечение следующего элемента списка

function TSetList.GetNext: TItem;
begin
  if Assigned(mCurrent)
    then mCurrent:= mCurrent^.mNext;
  if Assigned(mCurrent)
    then Result:= mCurrent^.mItem
    else Result:= nil;
end;

// Сохранение в стеке текущей позиции перебора

procedure TSetList.PositionPop;
begin
  mCurrent:= PListRec(mStack.Pop);
end;

// Извлечение из стека текущей позиции перебора

procedure TSetList.PositionPush;
begin
  if not Assigned(mStack)
    then mStack:= TBuffer.Create;
  mStack.Push(TItem(mCurrent));
end;

// Очистка множества без уничтожения элементов

procedure TSetList.Clr(arg: boolean);
var p : PListRec;
begin
  while Assigned(mHead) do begin
    p:= mHead;
    mHead:= mHead^.mNext;
    if arg and Assigned(p^.mItem)
       then p^.mItem.Free;
    Dispose(p);
  end;
  mCount:= 0;
end;
```

```
// Деструктор множества

destructor TSetList.Destroy;
begin
  mStack.Free;
  inherited;
end;

// Проверка наличия элемента в множестве

function TSetList.Exist(arg: TItem): boolean;
var p : PListRec;
begin
  Result:= false;
  if not Assigned(arg) then Exit;
  p:= mHead;
  while Assigned(p) do begin
    if p.mItem = arg then begin Result:= true; Break end;
    case p^.mItem.Compare(arg) of
      cmpLess,
      cmpIncomp: p:=p^.mNext;  // меньше или несравнимо
      cmpEq    : begin Result:= true; Break end;
      cmpGreate: Break;  // больше
    end;
  end;
end;

// Вспомогательная процедура вставки

procedure TSetList.Ins(p: PListRec; var q: PListRec; arg: TItem);
var r : PListRec;
begin
  Inc(mCount);
  New(r);  r^.mItem:= arg; r^.mNext:= p;
  if Assigned(q)
    then q^.mNext:= r
    else mHead:= r;
  q:= r;
end;

// Вставка элемента в множество
// Возвращает FALSE,
// если элемент или его эквивалент уже содержится в множестве

function TSetList.Insert(arg: TItem): boolean;
var p, q : PListRec;
    cmp : TCompare;
begin
  Result:= false;
  if not Assigned(arg) then Exit;
  cmp:= cmpEq;
  p:= mHead; q:=nil;
  while Assigned(p) do begin
    cmp:= p^.mItem.Compare(arg);
    if cmp in [cmpEq, cmpGreate] then Break;
    q:= p;  p:= p^.mNext
  end;
  if not Assigned(p) or (cmp = cmpGreate) then begin
    Ins(p,q,arg);
    Result:= true;
  end;
end;
```

```
// Вспомогательная процедура удаления

procedure TSetList.Del(var p, q: PListRec);
begin
  if Assigned(q)
    then q^.mNext:= p^.mNext
    else mHead:= p^.mNext;
  Dispose(p);
  if Assigned(q)
    then p:= q^.mNext
    else p:= mHead;
  Dec(mCount);
end;

// Процедура удаления элемента из множества

procedure TSetList.Delete(arg: TItem);
var p, q : PListRec;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
  while Assigned(p) do begin
    case p^.mItem.Compare(arg) of
      cmpLess: begin q:= p;  p:= p^.mNext;  end;
      cmpEq:   begin Del(p,q); Break; end;
      else Break;
    end;
  end;
end;

// Сложение множеств: аргумент добавляется к данному множеству

procedure TSetList.Add(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not (arg is TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpEq;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      case cmp of
        cmpLess  : begin q:= p; p:= p^.mNext end;
        cmpEq    : begin t:= arg.GetNext; break end;
        else  Break;
      end;
    end;
    if not Assigned(p) or (cmp = cmpGreate) then begin
      Ins(p,q,t);
      if Assigned(t) then t:= arg.GetNext;
    end;
  end;
  arg.PositionPop;
end;
```

```
// Вычитание множеств: аргумент вычитается из данного множества

procedure TSetList.Sub(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  if not arg.InheritsFrom(TSet) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpLess;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      if cmp in [cmpEq, cmpGreate] then Break;
      q:= p; p:= p^.mNext;
    end;
    if cmp = cmpEq then Del(p,q);
    if not Assigned(p) then Break;
    t:= arg.GetNext;
  end;
  arg.PositionPop;
end;

// Пересечение множеств: аргумент пересекается с данным множеством

procedure TSetList.Mul(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    cmp:= cmpEq;
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      if cmp <> cmpEq then Break;
      q:= p;
      p:= p^.mNext;
    end;
    if cmp = cmpLess then Del(p,q);
    if not Assigned(p) then Break;
    if cmp = cmpGreate then t:= arg.GetNext;
  end;
  while Assigned(p) do Del(p,q);
  arg.PositionPop;
end;

// Исключающее ИЛИ множеств: результат в данном множестве

procedure TSetList.ExOr(arg: TSet);
var p, q : PListRec;
    t : TItem;
    cmp : TCompare;
begin
  if not Assigned(arg) then Exit;
  p:= mHead; q:= nil;
```

```
  arg.PositionPush;
  t:= arg.GetFirst;
  while Assigned(t) do begin
    while Assigned(p) do begin
      cmp:= p^.mItem.Compare(t);
      case cmp of
        cmpLess:
          begin  q:= p; p:= p^.mNext end;
        cmpEq:
          begin
            Del(p, q);
            t:= arg.GetNext;
            if not Assigned(t) then break;
          end;
        cmpGreate:
          begin
            Ins(p, q, t);
            t:= arg.GetNext;
            if not Assigned(t) then break;
          end;
      end;
    end;
    while Assigned(t) do begin
      Ins(p, q, t);
      t:= arg.GetNext
    end;
  end;
  arg.PositionPop;
end;

end.
```

Приложение E
Модуль Dissect

```
unit Dissect;
//*************************************************************************
//               Разбиения множества на подмножества                      *
// Функции:                                                                *
// GenAllDissections -- Генерирует все разбиения множества                 *
// GenDissectRand -- Генерирует заданное количество разбиений              *
// GenSubsRand -- Генерирует заданное количество подмножеств               *
// GenSubsFromText -- Генерирует подмножества по текстовому файлу          *
//*************************************************************************

interface

uses Root, Items, SetList;

// Генерация всех разбиений множества aSet

function GenAllDissections(aSet: TSet): TSet;

// Генерирует заданное в aCnt количество разбиений
// исходного множества aSet.
// Параметры задают минимальную и максимальная мощность подмножеств,
// входящих в разбиения

function GenDissectRand(
                     aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TSet;

// Генерирует заданное в aCnt количество подмножеств
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
// Совокупность полученных подмножеств обладает двумя свойствами разбиения:
// - сумма всех подмножеств равна aSet
// -- произведение всех подмножеств пусто
// Но в отличие от разбиения некоторые подмножества совокупности
// могут пересекаться и даже совпадать.

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;

// Генерация из текстового файла
// набора подмножеств TBuffer и УНИВЕРСУМА aSet
// Используются только буквы 'a'..'z'

function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;
```

```
////////////////////////////////////////////////////////////////////////////
implementation
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Генерация всех разбиений множества ( Липский, стр 43 )
////////////////////////////////////////////////////////////////////////////

function GenAllDissections(aSet: TSet): TSet;
var Que    : TBuffer;  // буфер-очередь
    SSGet  : TSet;       // извлекаемое множество подмножеств
    SSPut : TSet;        // добавляемое множество подмножеств
    //---------------------------------------
    // Процедура обработка всех подмножеств
    // из множества множеств SSGet

    procedure LocalHandle(aItem: TItem);
    var k: integer;
        S: TSet;     // текущее множество из множества подмножеств
    begin
      // Обработка очередного извлечённого множества подмножеств SSGet,
      // перебираем его элементы-множества S:
      for k:= 1 to SSGet.GetCount do begin
          SSPut:= SSGet.Copy as TSet;   // копия извлеч. множества
подмножеств
          S:= SSPut.GetItem(k) as TSet; // очередное множество из МП
          SSPut.Delete(S);   // удаляем из него очередное подмножество
          S.Insert(aItem);   // к очередному добавляем текущий элемент
          SSPut.Insert(S);   // полученное вставляем в множество подмножеств
          Que.Put(SSPut);    // новое множество подмножеств заносим в очередь
      end;
      // Добавление нового подмножества к SSGet
      SSPut:= SSGet.Copy as TSet;  // копия извлеч. множества подмножеств
      S:= CreateSet;       // создаём множество
      S.Insert(aItem);     // из единственнного элемента aItem
      SSPut.Insert(S);     // добавляем к множеству подмножеств
      Que.Put(SSPut);      // новое множество подмножеств заносим в очередь
    end;
    //---------------------------------------
var   i, j  : integer;  // счётчики
      Item: TItem;      // текущий элемент исходного множества

begin
  Que:= TBuffer.Create; // создание буфера для очереди
  SSPut:= CreateSet;       // пустое множество подмножеств
  Que.Put(SSPut);          // заносим в очередь
  // Цикл по всем элементам исходного множества:
  for i:= 1 to aSet.GetCount do begin
    Item:= aSet.GetItem(i);
    // Цикл обработки очереди:
    for j:=1 to Que.GetCount do begin
      SSGet:= Que.Get as TSet;  // берём очередное множество из очереди
      LocalHandle(Item);        // создаём из него ряд других
      SSGet.ClrAndDestroy;      // очищаем от подмножеств
      SSGet.Free;               // и уничтожаем
    end;
  end;
  Result:= CreateSet; // создание пустого результата
  // Перенос из буфера в множество-результат
  while Que.GetCount>0 do Result.Insert(Que.Get);
  Que.Free;  // удаление буфера
end;
```

```
////////////////////////////////////////////////////////////////////////////
//  Генерация случайного разбиения исходного множества aSet
//  Внимание: aSet очищается!
////////////////////////////////////////////////////////////////////////////

function GenRandSet(aMin, aMax: integer; aSet: TSet): TSet;
var n, k, j: integer;
    t: TItem;
    S: TSet;
begin
  Result:= CreateSet;                 // создаём множество-результат
  repeat
    n:= aMin + Random(aMax-aMin);     // случайная мощность подмножества
    S:= CreateSet;                    // создаём подмножество
    for j:= 1 to n do begin           // и заполняем его случайным выбором эл-
в
      k:= 1 + Random(aSet.GetCount); // случайный индекс в исходном множестве
      t:= aSet.GetItem(k);            // случайный элемент исходного множества
      S.Insert(t);                    // вставляем в множество
      aSet.Delete(t);                 // и удаляем из исходного
      if aSet.GetCount=0 then Break; // выход из цикла, если исчерпано
    end;
    if S.GetCount>0           // если множество не пусто,
      then Result.Insert(S)   // то вставляем в результат
      else S.Free;            // а иначе удаляем
  until aSet.GetCount=0;      // пока не исчерпано исходное множество
end;

////////////////////////////////////////////////////////////////////////////
// Генерирует заданное в aCnt количество разбиений
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
////////////////////////////////////////////////////////////////////////////

function GenDissectRand(
                      aSet: TSet;  { исходное множество }
                      aCnt,  { количество генерируемых разбиений }
                      aMin,  { минимальная мощность подмножеств, %}
                      aMax   { максимальная мощность подмножеств, %}
                      : integer): TSet;
var
  nMin, nMax: Integer;
  Copy: TSet;    // рабочая копия исходного множества aSet
  SS: TSet;      // очередное множество подмножеств
  i: Integer;

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= CreateSet;
  for i:= 1 to aCnt do begin
    Copy:= aSet.Copy as TSet;
```

```
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);
      Result.Insert(SS);
    until Copy.GetCount=0;
    Copy.Free;
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Генерирует заданное в aCnt количество подмножеств
// исходного множества aSet.
// Параметры задают минимальную и максимальную (%) мощность подмножеств,
// входящих в разбиения
// В отличие от разбиения некоторые подмножества совокупности
// могут пересекаться и даже совпадать.
////////////////////////////////////////////////////////////////////////////

function GenSubsRand(aSet: TSet;  { исходное множество }
                     aCnt,  { количество генерируемых разбиений }
                     aMin,  { минимальная мощность подмножеств, %}
                     aMax   { максимальная мощность подмножеств, %}
                     : integer): TBuffer;
var
  nMin, nMax: Integer;
  Copy: TSet;  // рабочая копия исходного множества aSet
  SS: TSet;    // очередное разбиение
  S : TSet;    // очередное множество внутри разбиения

begin
  // Перевод процентов в количество

  nMin:= (aSet.GetCount * aMin) div 100;   // aMin %
  if nMin<1 then nMin:=1;
  if nMin>=aSet.GetCount then nMin:=aSet.GetCount-1;

  nMax:= (aSet.GetCount * aMax) div 100;   // aMax %
  if nMax<nMin then nMax:=nMin;
  if nMax>aSet.GetCount then nMax:=aSet.GetCount;

  Result:= TBuffer.Create;  // создаём буфер результата

  // Цикл создания заданного количества подмножеств

  while aCnt>0 do begin
    Copy:= aSet.Copy as TSet;   // создаём копию исходного множества
    repeat
      SS:= GenRandSet(nMin, nMax, Copy);  // создаём случайное разбиение
      // Перебор подмножеств случайного разбиения:
      S:= SS.GetFirst as TSet;   // первое подмножество из разбиения
      while Assigned(S) and (aCnt>0) do begin
        if Random(3)<>0 then begin
          // копия очередного "осколка"
          // с вероятностью 2/3 вставляется в результат
          Result.Put(S.Copy);
          Dec(aCnt);
        end;
        S:= SS.GetNext as TSet;  // следующее подмножество из разбиения
      end;
      SS.ClrAndDestroy;  // очищаем разбиение
      SS.Free;           // и удаляем его
    until (aCnt=0) or (Copy.GetCount=0);
    Copy.Free;   // удаляем копию исходного множества
```

```
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Генерация из текстового файла
// набора подмножеств TBuffer и УНИВЕРСУМА aSet
// Используются только буквы 'a'..'z'
////////////////////////////////////////////////////////////////////////////

function GenSubsFromText(const aName: String;  // имя файла
                         var aSet: TSet;       // результат-УНИВЕРСУМ
                         aCnt: integer         // количество подмножеств
                         ): TBuffer;

type TChrSet = set of char;

    // - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Формирование объекта-множества символов
    // из классического множества символов

    function MakeFromSet(const aSet: TChrSet): TSet;
    var c: char;
        t: TItemChar;
    begin
      Result:= CreateSet;
      for c in aSet do begin
        t:= TItemChar.Create(c);
        Result.Insert(t)
      end;
    end;
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var F: Text;            // текстовый файл
    S: string;          // строка файла
    c: char;            // очередной символ
    i: integer;         // индекс
    set1 : TChrSet;     // множество для УНИВЕРСУМа
    set2 : TChrSet;     // множество для подмножества
    subset: TSet;       // очередное подмножество

begin  { GenSubsFromText }

  Result:= TBuffer.Create;
  set1:= [];
  set2:= [];
  Assign(F, aName); Reset(F);
  while not Eof(F) do begin
    if Result.GetCount >= aCnt then break;
    Readln(F, S);
    for i:= 1 to Length(S) do begin
      // Обработка очередной строки файла:
      if Result.GetCount >= aCnt then break;
      c:= Char(Ord(S[i]) or $20);        // low case (буквы 'a'..'z')
      if c in ['a'..'z'] then begin
        set1:= set1 + [c];               // для УНИВЕРСУМА
        if c in set2 then begin          // если символ встретился повторно
           subset:= MakeFromSet(set2);   // создаём подмножество
           Result.Put(subset);           // и заносим в результат
           set2:= [c];                   // начинаем следующее подмножество
        end else begin                   // если символ встретился впервые
           set2:= set2 + [c];            // расширяем текущее подмножество
        end;
```

```
      end;
    end;
  end;
  Close(F);
  subset:= MakeFromSet(set2);   // оставшееся множество
  Result.Put(subset);           // вставляем в результат
  aSet:= MakeFromSet(set1);     // и создаём универсум
end;

end.
```

Приложение F
Модуль Assembly

```
unit Assembly;
{$I Common.inc}
//*************************************************************************
//                       Сборка множеств                                  *
// Функции:                                                               *
// CollectMinDissect -- Задача о наименьшем разбиениии (ЗНР)              *
// CollectMinCover   -- Задача о наименьшем покрытии (ЗНП)                *
// CollectGradCover  -- быстрый градиентный алгоритм для ЗНП              *
// Кристофидес стр. 53                                                    *
//*************************************************************************

interface

uses Root, Items, SetList;

//  Задача о наименьшем разбиениии (ЗНР)

function CollectMinDissect(aUniv: TSet;      // целевой универсум
                           aBuf: TBuffer   // буфер с подмножествами
                           ):TCostSet;

// Задача о наименьшем покрытии (ЗНП)
// Кристофидес стр. 53

function CollectMinCover(aUniv: TSet;      // целевой универсум
                         aBuf: TBuffer   // буфер с подмножествами
                        ):TCostSet;

// Градиентный алгоритм для задачи о наименьшем покрытии ЗНП

function CollectGradCover(aBase: TSet;      // целевой универсум
                          aBuf: TBuffer   // буфер с подмножествами
                         ):TCostSet;

///  Оценка трудоёмкости полного перебора множества блоков

function CalcDifficult(aSet: TSet): Extended;

////////////////////////////////////////////////////////////////////////////
implementation
uses SetUtils;

////////////////////////////////////////////////////////////////////////////
// TSetBlock
// Специальный тип множества для решения задач
// о минимальном разбиении (ЗНР) и покрытии (ЗНП)
////////////////////////////////////////////////////////////////////////////

type

  TSetBlock = class (TSetList)
    mLabel : TItem;    // Метка блока
    constructor Create(aLabel: TItem);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: Text); override;
  end;
```

```
constructor TSetBlock.Create(aLabel: TItem);
begin
  inherited Create;
  mLabel:= aLabel;
end;

function TSetBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;  // порядок элементов совпадает с порядком вставки
end;

procedure TSetBlock.Print(var aFile: Text);
begin
  mLabel.Print(aFile);
  inherited;
end;

////////////////////////////////////////////////////////////////////////////
//  Проверка множества подмножеств на предмет того,
//  можно ли составить разбиение или покрытие?
////////////////////////////////////////////////////////////////////////////

function TestCoverOrDissect(aUniv: TSet;     // универсум
                            aBuf: TBuffer  // буфер подмножеств
                           ): boolean;
var  CS : TCostSet;  // очередное подмножество с оценкой
    Sum : TSet;      // накопитель универсума
      i : integer;
begin
  Result:= false;
  Sum:= CreateSet;
  // Суммируем подмножества буфера до тех пор,
  // пока либо сумма достигнет универсума, либо исчерпается буфер
  for i:=1 to aBuf.GetCount do begin
    CS:= aBuf.GetByIndex(i) as TCostSet;
    Sum.Add(CS.mSet);
    if Sum.Compare(aUniv) in [cmpEq, cmpGreate] then begin
      Result:= True;
      Break;
    end;
  end;
  Sum.Free;
end;

////////////////////////////////////////////////////////////////////////////
///   М И Н И М А Л Ь Н О Е   Р А З Б И Е Н И Е  (DISSECTION)
////////////////////////////////////////////////////////////////////////////

// Просеивание дубликатов бОльшей или равной стоимости

function GenFilter(aBuf: TBuffer): TBuffer;
var Ci, Cj : TCostSet;
      i, j : integer;
      Comp : TCompare;
begin
  // Предварительный сброс флажков:
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    Ci.mFlag:= false;
  end;
  // Установка флажков для пропускаемых элементов
  // Отмечаем совпадающие подмножества с бОльшнй либо равной ценой
```

```
  for i:= 1 to aBuf.GetCount-1 do begin
    Ci:= aBuf.GetByIndex(i) as TCostSet;
    if Ci.mFlag then Continue;              // если обработан
    for j:= i+1 to aBuf.GetCount do begin
      Cj:= aBuf.GetByIndex(j) as TCostSet;
      if Cj.mFlag then Continue;            // если обработан
      Comp:= Ci.mSet.Compare(Cj.mSet);      // сравниваем подмножества
      if Comp = cmpEq then begin
        // Если сопадают, сравниваем цены:
        if Ci.mCost >= Cj.mCost
          then Ci.mFlag:= true    // i-й будет пропущен
          else Cj.mFlag:= true;   // j-й будет пропущен
      end;
    end;
  end;

  Result:= TBuffer.Create;
  // Перепись в буфер результата элементов со сброшенным флажком
  // Прокручиваем циклически исходный буфер aBuf
  for i:= 1 to aBuf.GetCount do begin
    Ci:= aBuf.Get as TCostSet;
    aBuf.Put(Ci);
    // Если флаг не установлен, то заносим в буфер результата
    if not Ci.mFlag  then Result.Put(Ci);
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Создаёт блоки подмножеств для поиска разбиений
// aSet  - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)
////////////////////////////////////////////////////////////////////////////

function GenDissectBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
var i, j: integer;
    t: TItem;          // очередной элемент из универсума
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet;                   // создаём множество-результат
  for i:=1 to aUniv.GetCount do begin  // цикл по элементам универсума
    if aBuf.GetCount=0 then Break;      // стоп при исчерпании буфера
    // Выбираем очередной элемент универсума:
    t:= aUniv.GetItem(i);
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.Get as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)        // если оно содержит текущий элемент
        then Block.Insert(S)    // то помещаем подмножество S в блок
        else aBuf.Put(S);       // а иначе возвращаем подмножество в буфер
    end;
    if Block.GetCount>0         // если блок не пуст
      then Result.Insert(Block) // помещаем его в множество
      else Block.Free;          // а иначе удаляем пустой
  end;
  aBuf.Free;  // освобождаем буфер
end;
```

```
////////////////////////////////////////////////////////////////////////////
//  Оценка трудоёмкости полного перебора множества блоков
////////////////////////////////////////////////////////////////////////////

function CalcDifficult(aSet: TSet): Extended;
var S: TSet;  // очередной блок
begin
  Result:=1;
  // Перемножаем мощности всех блоков (пустых нет)
  S:= aSet.GetFirst as TSet;
  while Assigned(S) do begin
    if S.GetCount=1
      then Result:= Result +1
      else Result:= Result * S.GetCount;
    S:= aSet.GetNext as TSet;
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Задача о наименьшем разбиениии ЗНР
// Кристофидес стр. 53
////////////////////////////////////////////////////////////////////////////

function CollectMinDissect(aUniv: TSet;      // целевой универсум
                           aBuf: TBuffer   // буфер с подмножествами
                           ): TCostSet;
var
    Buf     : TBuffer;  // Отфильтрованная копия входного буфера
    Univ    : TSet;       // Накопитель универсума
    BestCost: integer;    // Лучшая стоимость
    Blocks  : TSet;       // Вспомогательные блоки
    Res : TCostSet;       // Исходное пустое разбиение
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блока

  procedure BlockHandle(aBlock: integer;  // номер блока
                        aCost: integer;   // накопленная сумма
                        aSum: TSet;       // накопленные элементы
                        aRes: TCostSet    // накопленные подмножества
                        );
  var Block: TSetBlock; // текущий блок
      Sum : TSet;       // Накопленное множество элементов
      Res : TCostSet;   // Накопленное разбиение (множество подмножеств)
       CS : TCostSet;   // очередное оценённое подмножество из блока
  begin
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Если элемент текущего блока содержится в накопителе
    if aSum.Exist(Block.mLabel) then begin
      // то пропускаем этот блок и входим в следующий (если не последний)
      if aBlock < Blocks.GetCount then begin
        BlockHandle(aBlock+1, aCost, aSum, aRes) // вход в следующий блок
      end
    end else begin
      // перебор подмножеств текущего блока
      CS:= Block.GetFirst as TCostSet;
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        // Если сумма не превышена
        // и CS не пересекается с накопл. универсумом, то обрабатываем
        if ((aCost+CS.mCost) < BestCost) and
           not aSum.TestIntersect(CS.mSet) then begin
          // Пристраиваем к разбиению очередной блок:
```

```
          Sum:= aSum.Copy as TSet;     // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного разбиения
          Sum.Add(CS.mSet);     // накапливаем универсум
          Res.Insert(CS);       // пристраиваем к разбиению очередной блок
          // Проверяем, накоплен ли целевой универсум
          if Sum.GetCount = aUniv.GetCount then begin
            // Здесь найдено очередное разбиение.
            // Поскольку оно лучше прежнего,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;   // новая лучшая сумма
            Result.Free;                   // удаляем прежнее разбиение
            Result:= Res.Copy as TCostSet; // и создаём новое лучшее
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res)
          end; // else
          // Перед выбором следующего подмножества в блоке освобождаем:
          // возвращаем из стека предыдущие значения
          Res.Free;   // накопленное разбиение
          Sum.Free;   // накопленный универсум
        end; // if
        CS:= Block.GetNext as TCostSet;
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------
begin  { CollectMinDissect }

  Result:= TCostSet.CreateEmpty;  // Создаём пустое множество-разбиение

  // Проверяем возможность найти разбиение, иначе выход

  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf);  // Создаём фильтрованную копию входного буфера

  Blocks:= GenDissectBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  // Подготовка переменных:
  BestCost:= MaxInt;          // Лучшая сумма
  Univ := CreateSet;          // Накопитель универсума
  Res:= TCostSet.CreateEmpty; // Создаём пустое множество-разбиение

  // Рекурсивная обработка блоков:
  BlockHandle(1, 0, Univ, Res);

  // Очистка:
  Res.Free;                   // Исходное пустое разбиение
  Univ.Free;                  // Освобождаем накопитель
  Blocks.ClrAndDestroy;       // Удаляем блоки
  Blocks.Free;                // и множество блоков
end;

////////////////////////////////////////////////////////////////////////////
//    М И Н И М А Л Ь Н О Е   П О К Р Ы Т И Е  (COVER)
////////////////////////////////////////////////////////////////////////////

// Создаёт блоки подмножеств для поиска минимальных покрытий
// aUniv - универсум
// aBuf  - совокупность разных его подмножеств (оценённых)

function GenCoverBlocks(aUniv: TSet; aBuf: TBuffer): TSet;
```

```
var i, j: integer;
    t: TItem;          // очередной элемент из aSet
    S: TCostSet;       // подмножество с оценкой
    Block: TSetBlock; // блок (множество подмножеств)
begin
  Result:= CreateSet; // создаём буфер результата
  // Цикл по элементам универсума:
  for i:=1 to aUniv.GetCount do begin
    t:= aUniv.GetItem(i);         // выбираем очередной элемент универсума
    Block:= TSetBlock.Create(t); // создаём блок
    // просмотр рабочего буфера:
    for j:= 1 to aBuf.GetCount do begin
      S:= aBuf.GetByIndex(j) as TCostSet; // выбираем очередное подмножество
      if S.mSet.Exist(t)       // если оно содержит текущий элемент
        then Block.Insert(S)  // то помещаем подмножество S в блок
    end;
    Result.Insert(Block) // помещаем блок в результирующее множество
  end;
end;

////////////////////////////////////////////////////////////////////////////
// Задача о наименьшем покрытии (ЗНП)
// Кристофидес стр. 53
////////////////////////////////////////////////////////////////////////////

function CollectMinCover(aUniv: TSet;       // целевой универсум
                         aBuf: TBuffer    // буфер подмножеств с оценками
                        ): TCostSet;
var Buf: TBuffer;        // отфильтрованный входной буфер
    Univ    : TSet;        // Накопленный универсум
    BestCost: integer;     // Лучшая сумма
    Blocks  : TSet;        // Вспомогательные блоки
    Res : TCostSet;        // Исходное пустое покрытие
  //-------------------------------------------------------------
  // Рекурсивная процедура обработки блоков

  procedure BlockHandle(aBlock, aCost: integer; aSum: TSet; aRes: TCostSet);
  var Block : TSetBlock; // текущий блок
      Sum : TSet;         // Накопленный универсум
      Res : TCostSet;     // Накопленное покрытие (множество подмножеств)
       CS : TCostSet;     // очередное оценённое подмножество из блока
  begin
    // Выбираем блок из множества блоков
    Block:= Blocks.GetItem(aBlock) as TSetBlock;
    // Проверяем наличие метки текущего блока в накопителе универсума
    if aSum.Exist(Block.mLabel) then begin
      // Метка уже есть, пропускаем данный блок
      // и входим в следующий, если текущий не последний
      if aBlock<Blocks.GetCount then BlockHandle(aBlock+1,aCost,aSum,aRes)
    end else begin
      // Начало обработки очередного блока
      CS:= Block.GetFirst as TCostSet; // первое подмножество блока
      // Перебор подмножеств блока
      while Assigned(CS) do begin
        // Попытка прилепить к разбиению очередное подмножество
        if (aCost+CS.mCost) < BestCost then begin // сумма меньше ?
          Sum:= aSum.Copy as TSet;     // копия накопленного универсума
          Res:= aRes.Copy as TCostSet; // копия накопленного покрытия
          Sum.Add(CS.mSet);     // накапливаем универсум
          Res.Insert(CS);       // и покрытие
          // Проверяем, накоплен ли целевой универсум (покрытие):
          if Sum.GetCount = aUniv.GetCount then begin
```

```
            // Найдено очередное покрытие,
            // обновляем прежнее лучшее значения
            BestCost:= aCost + CS.mCost;    // новая лучшая сумма
            Result.Free;                    // удаляем прежнее покрытие
            Result:= Res.Copy as TCostSet; // и сохраняем копию нового
          end else begin
            // Накопитель универсума не полон, входим в следующий блок
            if aBlock < Blocks.GetCount
              then BlockHandle(aBlock+1, aCost + CS.mCost, Sum, Res);
          end; // else
          // Перед выбором следующего подмножества из блока удаляем:
          Res.Free;  // накопленное покрытие
          Sum.Free;  // накопленный универсум
        end; // then
        CS:= Block.GetNext as TCostSet;  // следующий элемент блока
      end; // while
    end; // else
  end;
  //-------------------------------------------------------------

begin  { CollectMinCover }

  Result:= TCostSet.CreateEmpty; // Создаём пустое покрытие

  // Проверяем, если покрытие недостижимо, то выход:
  if not TestCoverOrDissect(aUniv, aBuf) then Exit;

  Buf:= GenFilter(aBuf); // Создаём фильтрованную копию входного буфера
  Blocks:= GenCoverBlocks(aUniv, Buf);  // Создаём вспомогательные блоки
  Buf.Free;                             // Буфер уже не нужен, удаляем
  Res:= TCostSet.CreateEmpty; // Начальное пустое покрытие
  Univ := CreateSet;          // Накопитель элементов (универсум)
  BestCost:= MaxInt;          // Лучшая сумма
  BlockHandle(1, 0, Univ, Res);  // Рекурсивная обработка блоков
  // Очистка:
  Res.Free;             // Начальное пустое покрытие
  Univ.Free;            // удаляем накопитель элементов (универсум)
  Blocks.ClrAndDestroy; // удаляем блоки
  Blocks.Free;
end;

////////////////////////////////////////////////////////////////////////////
// Градиентный алгоритм задачи о наименьшем покрытии ЗНП
////////////////////////////////////////////////////////////////////////////

function CollectGradCover(aBase: TSet;
                          aBuf: TBuffer
                         ): TCostSet;
var    Buf : TBuffer;    // отфильтрованный входной буфер
      Univ : TSet;       // накопитель универсума
    Blocks : TSet;       // блоки
     Block : TSetBlock;  // текущий блок
    BestCS : TCostSet;   // текущее лучшее покрытие
         i : Integer;
  //-------------------------------------------------------------
  function GetBest: TCostSet;
  var      CS : TCostSet;   // очередное подмножество с оценкой
            S : TSet;       // рабочее подмножество
    BestDelta : integer;    // лучшее приращение
     BestCost : integer;    // лучшая цена
            i : integer;
  begin
```

```
    Result:= nil;
    S:= CreateSet;  // создаём рабочее множество
    BestDelta:=0;   // лучшая мощность
    BestCost:= 1;   // лучшая цена
    // перебор элементов блока:
    for i:= 1 to Block.GetCount do begin
      CS:= Block.GetItem(i) as TCostSet;  // очередное подмнож. с оценкой
      S.CopyItems(CS.mSet);               // копируем подмножество
      S.Sub(Univ);                        // оставляем приращение множества
      // Если данное приращение лучшее, то запоминаем
      if S.GetCount * BestCost > CS.mCost * BestDelta then begin
        Result:= CS;             // лучшее множество
        BestCost:= CS.mCost;     // лучшая цена
        BestDelta:= S.GetCount;  // лучшее приращение
      end;
    end;
    S.Free;  // удаляем рабочее множество
  end;
  //-------------------------------------------------------------

begin  { CollectGradCover }

  Result:= TCostSet.CreateEmpty;  // Покрытие = пустое множество

  // Если покрытие невозможно, вернуть пустой результат

  if not TestCoverOrDissect(aBase, aBuf) then Exit;

  Buf:= GenFilter(aBuf);// Отфильтрованная копия входного буфера
  Blocks:= GenCoverBlocks(aBase, Buf);  // Вспомогательные блоки
  Buf.Free;             // Буфер уже не нужен
  Univ:= CreateSet;     // Создаём накопитель универсума
  // Цикл обработки блоков:
  for i:= 1 to Blocks.GetCount do begin
    Block:= Blocks.GetItem(i) as TSetBlock;
    if Univ.Exist(Block.mLabel)  // если метка блока найдена в накопителе
      then Continue;             // то пропускаем блок
    BestCS:= GetBest;            // выбор лучшего подмножества в блоке
    if Assigned(BestCS) then begin
      // Если лучшее подмножество в блоке найдено:
      Univ.Add(BestCS.mSet);     // накапливаем универсум
      Result.Insert(BestCS);     // и покрытие
    end;
  end;
  // Очистка памяти:
  Univ.Free;              // освободить накопитель универсума
  Blocks.ClrAndDestroy;   // очистить вспомогательные блоки
  Blocks.Free;
end;

end.
```

Приложение G
Модуль Graph

```
unit Graph;
{$I Common.inc}
//*************************************************************************
//                           Базовый ГРАФ                                 *
//                                                                        *
// Содержит объекты:                                                      *
//                                                                        *
// TNode -- базовая вершина (узел)                                        *
// TNodeInt -- вершина, помеченная числом                                 *
// TCondensNode -- вершина конденсата                                     *
// TLink -- базовый линк                                                  *
// TFarLink -- дальний указатель линк                                     *
// TGraph -- базовый граф                                                 *
// THugeGraph -- огромный граф с вершинами-числами                        *
// TCostNodes -- множество вершин с оценкой                               *
// TSortedNum -- самосортирующиеся числа                                  *
// TGamma -- множество вершин                                             *
// TAreal -- объект для поиска центров                                    *
// TTreeLink -- объект для связывания поддеревьев                         *
// TPair -- пара вершин для поиска паросочетаний                          *
// TPairsBlock -- блок пар для поиска паросочетаний                       *
// THamNode -- вершина вспомогательного графа для задачи Гамильтона       *
// THamLink -- дуга вспомогательного графа для задачи Гамильтона          *
// THamGraph -- вспомогательный граф для задачи Гамильтона                *
// TClosed -- множество закрытых линков задачи Гамильтона                 *
//*************************************************************************

interface

uses Root, Items;

type

  TReal = Single;  // данные действительного типа

  // Варианты карты дальних указателей при построении центров и медиан:

  TCenter = ( Undefined,  // не определена
              InCenter,   // внутренний центр и медиана
              OutCenter,  // внешний центр и медиана
              InOutCenter // внешне-внутренний центр и медиана
            );

  // Варианты формирования паросочетаний:

  TPairs = (pMaxN,  // максимальной мощности
            pMaxW,  // максимального веса
            pMinW   // минимального веса
           );

  // Предварительные объявления:

  TNode = class;      // вершина
  TLink = class;      // связь
  TGraph = class;     // граф
  TFarLink = class;   // дальний указатель
```

```
  // Базовая вершина
  TNode = class (TItem)       // Вершина (узел) графа
    private
      mLnkOut : TSet;         // Множество исходящих связей
      mLnkIn  : TSet;         // Множество входящих связей
      mFarLinks: TSet;        // Множество дальних связей для карты
      function GetNear(aFar: TNode): TNode;
      function GetFarLink(aFar: TNode): TFarLink;
      procedure SortFarLinks;
      procedure PrintFarLinks(var aFile: TextFile);
      procedure ExpoFarLinks;
      procedure ExpoData;
    protected
      mOwner  : TGraph;       // Ссылка на содержащий узел граф
      mColor  : integer;      // Цвет
      mDist   : integer;      // Расстояние (Дейкстра)
                              // Текущий штраф (цепь Гамильтона)
      mPred   : TNode;        // Предшествующий узел (Дейкстра)
      mLink   : TLink;        // Связывающая дуга в цепочке (потоки)
      mRoot   : TItem;        // Корень (остовные деревья, цепь Гамильтона)
      mFlow   : integer;      // Допустимый поток (потоки)
                              // Приращение штрафа (цепь Гамильтона)
      mLimit  : integer;      // Предел посещаемости (задача почтальона)
      mPower  : integer;      // Текущая степень вершины (цепь Гамильтона)
      function  MakeLink(aDest: TNode; aVal : integer): TLink;
      function  GetLink(aDest: TNode): TLink;
      procedure RemoveLink(aDest: TNode);
      procedure AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
      procedure SetLinksLimit(aLimit: integer);
      function  IsLeft: boolean;
      function  IsRight: boolean;
    public
      mValue : integer;       // Вес узла
      constructor Create(aVal: integer; aOwner: TGraph);
      destructor Destroy;  override;
      function  Compare(arg: TItem): TCompare; override;
      function  GetName: string;  virtual;
      procedure ResetNode;
      procedure PrintLinks(var aFile: TextFile);
      procedure Print(var aFile: TextFile); override;
      function  GenStrongArea: TSet;
      // вспомогательные функции для работы с исходящими связями
      function  OutGetCnt: integer;
      function  OutLinkFirst: TLink;
      function  OutLinkNext: TLink;
      procedure OutPosPush;
      procedure OutPosPop;
      function  OutGetLink(aIndex: integer): TLink;
      procedure OutGammaAdd(aRes : TSet);
      procedure OutGammaGet(aRes : TSet);
      function  CalcCost: integer;
      function  GenGammaOut: TSet;
      // вспомогательные функции для работы с входящими связями
      function  InGetCnt: integer;
      function  InLinkFirst: TLink;
      function  InLinkNext: TLink;
      procedure InPosPush;
      procedure InPosPop;
      function  InGetLink(aIndex: integer): TLink;
      procedure InGammaAdd(aRes : TSet);
      function  GenGammaIn: TSet;
  end;
```

```
// Вершина, помеченная числом

TNodeInt = class (TNode)
  public
    mNumber  : integer;  // хранимое число
    constructor Create(aNumber, aVal: integer; aOwner: TGraph);
    function Compare(arg: TItem): TCompare; override;
    function Copy: TItem; override;
    function GetName: string;  override;
    function GetLinkByNumber(aNumber : integer): TLink;
end;

// Линк -- ребро или дуга

TLink = class (TItem)
  protected
    mLoadLinks : boolean;  // Признак нагруженных связей
    mLimit  : integer;     // Предел посещаемости (циклы)
  public
    mValue: integer;  // вес (стоимость, расстояние) дуги
    mOwner: TNode;    // источник дуги (владелец)
    mDest : TNode;    // приёмник дуги
    mColor: integer;  // цвет дуги (паросочетания)
    // Для решения потоковых задач:
    mLow   : integer; // допустимые минимальный поток  mLow <= mHigh
    mHigh  : integer; // допустимые максимальный поток
    mFlow  : integer; // текущий поток
    mTemp  : integer; // для временного хранения mHigh
    mDirect: integer; // направление дуги (+1 -- прямая, -1 -- обратная)
    constructor Create(aOwner, aDest: TNode; aVal: integer;
                       aLoadLinks: boolean);
    function  Compare(arg: TItem): TCompare; override;
    function  Copy: TItem; override;
    function  GetReverse: TLink; // поиск встречного линка
    procedure Print(var aFile: TextFile); override;
    procedure Print2(var aFile: TextFile);
end;

// Базовый граф

TGraph = class (TItem)
  private
    mInitMap : boolean; // Признак инициализации карты дальних связей
    // Для поиска p-центров:
    mMapDirect : TCenter; // направление для карты дальних связей
  protected
    mNodes : TSet;      // Множество вершин
  public
    mName : String;     // Произвольное имя
    mDirect : boolean;  // Признак орграфа
    mLoadNodes : boolean;  // Признак нагруженных вершин
    mLoadLinks : boolean;  // Признак нагруженных связей
  protected
  // Вспомогательные методы
    procedure PosPush;
    procedure PosPop;
    procedure ResetNodes;                    // очистка вершин
    function  GetNode(aNode: TNode): TNode;  // поиск вершин по полям
    procedure SetLink(aSource, aDest : TNode; aVal : integer);
    procedure SetLinksLimit(aLimit: integer);
    procedure Compress; // Замена длин дуг кратчайшими расстояниями
  public
```

```
    // Базовые методы
      constructor Create(const aName: string; aDir, aLNodes, aLLinks :
boolean);
      destructor Destroy;  override;
      function  InsertNode(aNode: TNode): boolean;
      procedure RemoveNode(aNode: TNode);  // удаление вершины из графа
      function  Compare(arg: TItem): TCompare; override;
      procedure Print(var aFile: TextFile); override;
      procedure Save(const aName: String); virtual; abstract;
      function  Nodes: integer;
      procedure ExpoFarLinks;
      procedure ExpoLinksData;
    // Перебор вершин
      function  NodeFirst: TNode;
      function  NodeNext: TNode;
    // Копирование и порождение подграфов
      function  CopyGr(aRevers: boolean): TGraph;
      function  Copy: TItem; override;
      function  CopyByNodes(aNodes: TSet): TGraph;
      function  CopyByLinks(aLinks: TSet): TGraph;
    // Достижимость
      function  TestLink(arg1, arg2 : TNode): boolean;
      function  CalcSteps(arg1, arg2 : TNode): integer;
      function  GenPath(aSource, aDest : TNode): TBuffer;
      function  GenDijkstra(aSource, aDest : TNode): TBuffer;
      function  GenQuickPath(aSource, aDest : TNode): TBuffer;
      function  GetQuickPathStr(aSource, aDest : TNode): String;
    private
      procedure InitMap_Floyd; // построение карты маршрутов
    public
    // Проверка связности (граф и орграф)
      function IsLinked: boolean;
    // Области связности в неориентированом графе
      function  CalcAreas: integer;  // подсчёт областей
      function  GenAreas: TSet;      // генерация областей связности
    // Достижимость и связность в ориентированом графе
      function  GenStrongAreas: TSet;
      function  GenCondens: TGraph;
    // Независимые множества, клики, доминирующие множества
      function  GenUndepend: TSet;
      function  GenClique: TSet;
      function  GenDominating: TCostSet;
    // Раскраска
      function  GenPaints(aQuick: boolean): TSet;
    // Центры
      function  GenCenter(var aLambda: integer; aDirect: TCenter): TSet;
      function  GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
      function  GenPCenters(aPoly: integer;
                            var aLambda: integer;
                            aCenter: TCenter): TCostSet;
    private
      procedure InitMap(aDirect: TCenter);
      procedure DoneMap;
      function  GenIntersections(aGams: TSet): TBuffer;
      function  GenLimitGams(aLambda: integer): TSet;
      function  GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
    // P-Медианы
    public
      // Поиск полным перебором
      function GenPMedian_A (aPoly: integer;    // кратность (P)
                             aDirect: TCenter;  // внешняя или внутр. медиана
                             var aCost: integer  // стоимость (результат)
```

```
                          ): TSet;
    // Поиск частичным перебором
    function GenPMedian (aPoly: integer;      // кратность (P)
                         aDirect: TCenter;    // внешняя или внутр. медиана
                         var aCost: integer   // стоимость (результат)
                        ): TSet;
  private
    function MedianCost: integer;      // Вычисление стоимости медианы
    procedure MedianCopy(aRes: TSet); // Копирование медианы в множество
  // Покрывающие деревья
  public
    function GenCoverTree(aMaxTree: boolean; var aCost: integer): TSet;
    function GenCoverDir(var aRoot: TNode; var aCost: integer): TSet;
    procedure ExpoAllCovers;
  // Потоки
  public
    function CalcMaxFlow0(aSource, aDest : TNode): integer;
    function CalcMinFlow (aSource, aDest : TNode): integer;
    function CalcMaxFlow (aSource, aDest : TNode): integer;
    function CalcMinCostFlow (aSource, aDest : TNode;
                              aFlow: integer ): integer;
  // Паросочетания
  public
    function GenPairs(aMode: TPairs): TCostSet;
    function MarkMinPairsDicoty(var aPairs: integer): integer;
    function GenMinPairsDicoty: TCostSet;
    function GenMinPairsDicotyFlow: TCostSet;
  // Задача почтальона
  protected
    function GenPostPathUndir(var aCost: integer): TBuffer;
    function GenPostPathDir(var aCost: integer): TBuffer;
  public
    function GenPostPath(var aCost: integer): TBuffer;
  public
  // Цепь Гамильтона
    function GenHamPath(aStart, aFin: TNode; var aCost: integer): TBuffer;
    function GenHamPathDir(aStart, aFin: TNode;
                           \ var aCost: integer): TBuffer;
    function GenHamPathStrip(aStart, aFin, aStrip: TNode;
                             var aCost: integer): TBuffer;
    function GenHamPath_Full(aStart, aFin: TNode;
                             var aCost: integer): TBuffer;
    function GenHamPath_Greed(aStart, aFin: TNode;
                              var aCost: integer;
                              aGreed: boolean): TBuffer;

  // Задача коммивояжёра и цикл Гамильтона
    function GenHamilton_Full(var aCost: integer): TBuffer;
    function GenHamilton_Greed(var aCost: integer;
                               aGreed: boolean): TBuffer;
    function GenHamilton(var aCost: integer): TBuffer;
    function GenHamilton_Quick(var aCost: integer): TBuffer;
end;

// Огромный Граф , вершины помечены числами

THugeGraph = class (TGraph)
  private
    function  GetLinkByNumber(aSource, aDest : integer): TLink;
    procedure MakeLink(aSource, aDest, aVal : integer);
  public
    constructor GenRandom(aDir: boolean;          // направленность
```

```
                              aLoadNodes, aLoadLinks, // нагруженность
                              aNodes,                 // количество вершин
                              aLinks                  // плотность связей,%
                              : integer);
      constructor GenFull(aDir: boolean;             // направленность
                          aLoadNodes, aLoadLinks,    // нагруженность
                          aNodes: integer);          // количество вершин
      constructor GenDicoty(aLoadLinks,              // нагруженность дуг
                              aPairs: integer);      // количество пар вершин
      function  GetNode(aNumber: integer): TNodeInt;
  end;


  // Дальний указатель для карты дальних связей

  TFarLink = class(TItem)    // Дальний указатель
    mNodeFar : TNode;        // целевая вершина
    mNodeNear: TNode;        // ближайшая вершина на пути к целевой
    mDist    : integer;      // расстояние от текущей к целевой
    mStep    : integer;      // цикл, на котором обрабатывается указатель
    constructor Create(aNear, aFar: TNode; aDist: integer);
    function Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile);  override;
  end;

  // Узел для создания конденсата (сильной компоненты)

  TCondensNode = class (TNode)
    public
      mName  : string;// имя = сумма входящих в компоненту имён вершин
      mNodes : TSet;  // собственные узлы
      mOut   : TSet;  // ссылки за пределы сильной компоненты (узлы)
      constructor Create(aNodes: TSet; aOwner: TGraph);
      destructor Destroy;  override;
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;

  // Элемент для представления доминирующих множеств

  TCostNodes = class (TCostSet)
    mNode: TNode;  // вершина-элемент доминирующего множества
    constructor Create(aCost: integer; aSet: TSet; aNode: TNode);
    procedure   Print(var aFile: TextFile); override;
  end;

  // Функция для создания огромного графа

function GenHugeGraph (aDir: boolean;           // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;

////////////////////////////////////////////////////////////////////////////
implementation

uses Assembly, SetList, SetUtils, SysUtils, DateUtils;
////////////////////////////////////////////////////////////////////////////

const              // "цвета" вершин:
    CWhite = 0;    // белая (не тронута)
```

```
    CGray  = 1;   // серая
    CBlack = 2;   // чёрная
    CRed   = 3;   // красная

    CInfinity = MaxInt;  // бесконечность

var _Trace: boolean = false;  // для отладки

////////////////////////////////////////////////////////////////////////////
// Выбор минимального из двух чисел
////////////////////////////////////////////////////////////////////////////

function Minimum(a1, a2 : integer): integer;
begin
  if a1 < a2 then Result:= a1 else Result:= a2;
end;

////////////////////////////////////////////////////////////////////////////
// Вспомогательная функция для преобразования
// множества вершин в строку символов
////////////////////////////////////////////////////////////////////////////

function NodesToStr(aSet: TSet): string;
var N: TNode;
begin
  Result:= '';
  N:= aSet.GetFirst as TNode;
  while Assigned(N) do begin
    if Length(Result)<>0 then Result:= Result+',';
    Result:= Result + N.GetName;
    N:= aSet.GetNext as TNode;
  end;
end;

////////////////////////////////////////////////////////////////////////////
// TNode -- вершина (узел) графа
////////////////////////////////////////////////////////////////////////////

constructor TNode.Create(aVal: integer; aOwner: TGraph);
begin
  inherited Create;             // унаследованный конструктор
  mOwner:= aOwner;              // владелец узла (граф)
  if mOwner.mLoadNodes          // если вершины нагружены
    then mValue:= aVal;         // то запомнить вес (по умолчанию = 0)
  mLnkOut:= CreateSet;          // пустое множество исходящих связей
  if mOwner.mDirect             // если орграф
    then mLnkIn := CreateSet;   // то создать пустое множество входящих связей
end;

destructor TNode.Destroy;
begin
  // Очистка и уничтожение:
  mLnkOut.ClrAndDestroy;    // исходящие линки
  mLnkOut.Free;             // множество исходящих линков
  mLnkIn.Free;              // множество входящих линков
  // Дальние указатели:
  if Assigned(mFarLinks) then begin
    mFarLinks.ClrAndDestroy;
    mFarLinks.Free;
  end;
  inherited;
end;
```

```
// Сравнение вершин:

function TNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as TNode).GetName
    then Result:= cmpLess
  else if GetName > (arg as TNode).GetName
    then Result:= cmpGreate
end;

// Имени у абстрактного узла не существует, поэтому формируем случайно

function TNode.GetName: string;
var N: integer;
begin
  N:= integer(Self) mod 99;
  Result:='x'+Char(Ord('0')+N div 10)+Char(Ord('0')+N mod 10);
end;

// Установка степеней для всех рёбер (исходящих дуг)

procedure TNode.SetLinksLimit(aLimit: integer);
var L : TLink;
begin
  L:= OutLinkFirst;
  while Assigned(L) do begin
    L.mLimit:= aLimit;
    L:= OutLinkNext;
  end;
end;

// Поиск дальней связи для построения маршрутной карты

function TNode.GetFarLink(aFar: TNode): TFarLink;
var  FL : TFarLink; // элемент дальней связи
begin
  Result:= nil;
  if not Assigned(mFarLinks) then Exit;
  mFarLinks.PositionPush;
  FL:= mFarLinks.GetFirst as TFarLink;    // первый элемент
  while Assigned(FL) do begin             // перебор
    if FL.mNodeFar = aFar then begin      // если нашли
      Result:= FL;                        // вернуть результат
      Break;
    end;
    FL:= mFarLinks.GetNext as TFarLink;  // следующий элемент
  end;
  mFarLinks.PositionPop;
end;

// Поиск ближайшего узла при построении маршрута по карте

function TNode.GetNear(aFar: TNode): TNode;
var  FL : TFarLink; // элемент дальней связи
begin
  Result:= nil;
  if not Assigned(mFarLinks) then Exit;
  mFarLinks.PositionPush;
  FL:= mFarLinks.GetFirst as TFarLink;    // первый элемент
```

```
  while Assigned(FL) do begin             // перебор
    if FL.mNodeFar = aFar then begin      // если нашли
      Result:= FL.mNodeNear;              // вернуть результат
      Break;
    end;
    FL:= mFarLinks.GetNext as TFarLink;   // следующий элемент
  end;
  mFarLinks.PositionPop;
end;

// Сортировка дальних указателей по неубыванию дистанции

procedure TNode.SortFarLinks;
var NewList : TSet;
begin
  if not Assigned(mFarLinks) then Exit;
  NewList:= CreateSet;
  NewList.CopyItems(mFarLinks);
  mFarLinks.Free;
  mFarLinks:= NewList;
end;

// Сброс узла перед поиском кратчайших путей, потоков и др.

procedure TNode.ResetNode;
begin
  mColor:= 0;       // цвет
  mPred:= nil;      // ссылка на предшествующий узел
  mLink:= nil;      // связывающая дуга в цепочке потока
  mDist:= MaxInt;   // расстояние от исходного узла
  mFlow:= MaxInt;   // допустимый поток
end;

// Установка связи между данным узлом и заданным
// aDest - вершина-приёмник
// aVal  - "вес" связи

function TNode.MakeLink(aDest: TNode; aVal: integer): TLink;
begin
  Result:= nil;
  if not Assigned(aDest) then Exit;
  Result:= GetLink(aDest);              // уже существует?
  if Assigned(Result) then Exit;        // да, выход
  // нет, создаём новую
  Result:= TLink.Create(Self, aDest, aVal, mOwner.mLoadLinks);
  // и вставляем исходящую связь
  if not mLnkOut.Insert(Result) then begin
    Result.Free;
    Result:= nil;
    Exit;
  end;
  if mOwner.mDirect                     // если орграф
    then aDest.mLnkIn.Insert(Result); // то вставляем в приёмнике как
входящую
end;

// Удаление связи между вершинами

procedure TNode.RemoveLink(aDest: TNode);
var  L: TLink;
begin
  if not Assigned(aDest) then Exit;
```

```
  OutPosPush;
  L:= OutLinkFirst;
  while Assigned(L) and (L.mDest <> aDest) do  L:= OutLinkNext;
  OutPosPop;
  if not Assigned(L) then Exit;
  mLnkOut.Delete(L);
  if mOwner.mDirect               // если орграф
    then aDest.mLnkIn.Delete(L); // то удаляем в приёмнике как входящую
  L.Free;
end;

// Поиск линка к указанной вершине

function TNode.GetLink(aDest: TNode): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) and (Result.mDest <> aDest)
    do  Result:= OutLinkNext;
  OutPosPop;
end;

// Создание множества сильной компоненты
// или связной области для графа
// Кристофидес, стр. 33

function TNode.GenStrongArea: TSet;
var Temp: TSet;
begin
  Result:= GenGammaOut;               // множество достижимых вершин
  if not mOwner.mDirect then Exit;    // выход, если это не орграф
  Temp:= GenGammaIn;                  // множество, из которых достижима
  Result.Mul(Temp);                   // пересечение множеств
  Temp.Free;                          // освобождаем
end;


// Вспомогательные методы для работы с исходящими связями

// Извлечение первого исходящего линка

function TNode.OutLinkFirst: TLink;
begin
  Result:= mLnkOut.GetFirst as TLink;
end;

// Извлечение следующего исходящего линка

function TNode.OutLinkNext: TLink;
begin
  Result:= mLnkOut.GetNext as TLink;
end;

// Сохранение позиции перебора

procedure TNode.OutPosPush;
begin
  mLnkOut.PositionPush;
end;

// Восстановление позиции перебора
```

```
procedure TNode.OutPosPop;
begin
  mLnkOut.PositionPop;
end;

// Получение исходящей связи по индексу

function TNode.OutGetLink(aIndex: integer): TLink;
begin
  Result:= TLink(mLnkOut.GetItem(aIndex));
end;

// Кол-во исходящих связей

function TNode.OutGetCnt: integer;
begin
  Result:= 0;
  if Assigned(mLnkOut) then Result:= mLnkOut.GetCount;
end;

// Накопление соседних узлов в множестве aSet

procedure TNode.OutGammaAdd(aRes : TSet);
var L: TLink;
begin
  L:= OutLinkFirst;            // первая исходящая связь
  while Assigned(L) do begin   // пока существуют
    aRes.Insert(L.mDest);      // вставить целевую вершину в результат
    L:= OutLinkNext;           // следующая исходящая связь
  end;
end;

// Получение исходящей гаммы

procedure TNode.OutGammaGet(aRes: TSet);
begin
  aRes.Clear;
  OutGammaAdd(aRes);
end;

// Создание множества вершин, достижимых из данной

function TNode.GenGammaOut: TSet;
var Node: TNode;
    Temp: TSet;
begin
  Result:= CreateSet;
  Result.Insert(Self);                // начианем расширение гаммы с текущей
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);           // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;    // начало перебора вершин
    while Assigned(Node) do begin
      Node.OutGammaAdd(Result);       // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;

// Подсчёт стоимости узла для поиска минимальных покрытий
```

```
function TNode.CalcCost: integer;
var L: TLink;
begin
  Result:=1;
  if mOwner.mLoadNodes then Result:= mValue;
  if not mOwner.mLoadLinks then Exit;
  // добавляем суммарную стоимость исходящих дуг (рёбер)
  L:= OutLinkFirst;
  while Assigned(L) do begin
    Inc(Result, L.mValue);
    L:= OutLinkNext;
  end;
end;

// Вспомогательные методы для работы с входящими связями

// Извлечение первого входящего линка

function TNode.InLinkFirst: TLink;
begin
  if mOwner.mDirect
    then Result:= mLnkIn.GetFirst as TLink
    else Result:= OutLinkFirst
end;

// Извлечение следующего входящего линка

function TNode.InLinkNext: TLink;
begin
  if mOwner.mDirect
    then Result:= mLnkIn.GetNext as TLink
    else Result:= OutLinkNext
end;

// Сохранение позиции перебора

procedure TNode.InPosPush;
begin
  if mOwner.mDirect
    then mLnkIn.PositionPush
    else OutPosPush
end;

// Восстановление позиции перебора

procedure TNode.InPosPop;
begin
  if mOwner.mDirect
    then mLnkIn.PositionPop
    else OutPosPop
end;

// Получение входящей связи по индексу

function TNode.InGetLink(aIndex: integer): TLink;
begin
  if mOwner.mDirect
    then Result:= TLink(mLnkIn.GetItem(aIndex))
    else Result:= OutGetLink(aIndex);
end;

// Накопление соседних узлов в множестве aSet
```

```
procedure TNode.InGammaAdd(aRes: TSet);
var L: TLink;
begin
  L:= InLinkFirst;
  while Assigned(L) do begin
    aRes.Insert(L.mOwner);
    L:= InLinkNext;
  end;
end;

// Количество входящих связей

function TNode.InGetCnt: integer;
begin
  Result:= 0;
  if mOwner.mDirect
    then begin
      if Assigned(mLnkIn) then Result:= mLnkIn.GetCount
    end
    else Result:= OutGetCnt
end;

// Возвращает TRUE, если вершина относится к левой доле
// направленного двудольного графа

function TNode.IsLeft: boolean;
begin
  Result:= mLnkOut.GetCount <> 0;
end;

// Возвращает TRUE, если вершина относится к правой доле
// направленного двудольного графа

function TNode.IsRight: boolean;
begin
  Result:= mLnkIn.GetCount <> 0;
end;

// Вывод в файл (на экран) исходящих связей вершины

procedure TNode.PrintLinks(var aFile: TextFile);
begin
  if Assigned(mLnkOut) then mLnkOut.Print(aFile);
end;

// Вывод в файл (на экран) вершины

procedure TNode.Print(var aFile: TextFile);
begin
  Write(aFile, GetName+' ');
  if mOwner.mLoadNodes then Write(aFile,'= ', mValue:2);
  Write(aFile, ' -> ');
  PrintLinks(aFile);
end;

// Вывод в файл (на экран) списка дальних связей

procedure TNode.PrintFarLinks(var aFile: TextFile);
var FL: TFarLink;
begin
  Print(aFile);
```

```
  with mFarLinks do begin
    PositionPush;
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      FL.Print(aFile);
      FL:= GetNext as TFarLink;
    end;
    PositionPop;
  end;
end;

// Отображение дополнительных данных (для отладки)

procedure TNode.ExpoData;
var L: TLink;
    Flag: boolean;
begin
  OutPosPush;
  Flag:= false;
  // Исходящие связи:
  L:= OutLinkFirst;
  while Assigned(L) do begin
    with L do begin
      Write(mOwner.GetName+'->'+mDest.GetName, '  C=', mValue:2);
      // Данные о потоке:
      Writeln('  L=', mLow:2,  '  H=', mHigh:2,
              '  F=', mFlow:2, '  T=', mTemp:2);
    end;
    Flag:= true;
    L:= OutLinkNext;
  end;
  if Flag then Writeln;
  OutPosPop;
end;

// Отображение дальних указателей (для отладки)

procedure TNode.ExpoFarLinks;
begin
  PrintFarLinks(Output);
end;

// Создание множества вершин, из которых достижима данная

function TNode.GenGammaIn: TSet;
var Node: TNode;
    Temp: TSet;
begin
  if not mOwner.mDirect then begin
    // Для неорграфа входящая гамма совпадает с исходящей
    Result:= GenGammaOut;
    Exit;
  end;
  Result:= CreateSet;
  Result.Insert(Self);                // начианем расширение гаммы с текущей
  Temp:= CreateSet;
  repeat
    Temp.CopyItems(Result);           // копирование текущей гаммы
    Node:= Temp.GetFirst as TNode;    // начало перебора вершин
    while Assigned(Node) do begin
      Node.InGammaAdd(Result);        // добавляем смежные вершины
      Node:= Temp.GetNext as TNode;
```

```
    end;
  until Result.GetCount = Temp.GetCount;  // пока гамма расширяется
  Temp.Free;
end;

////////////////////////////////////////////////////////////////////////////
// TNodeInt -- Вершина, помеченная числом
////////////////////////////////////////////////////////////////////////////

constructor TNodeInt.Create(aNumber, aVal: integer; aOwner: TGraph);
begin
  inherited Create(aVal, aOwner);
  mNumber:= aNumber;
end;


function TNodeInt.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  Result:= cmpLess;
  if (arg is TNodeInt) and (mNumber > (arg as TNodeInt).mNumber)
    then Result:= cmpGreate
end;


function TNodeInt.Copy: TItem;
begin
  Result:= TNodeInt.Create(mNumber, mValue, mOwner);
end;

// Получение линка на соседнюю вершину по номеру

function TNodeInt.GetLinkByNumber(aNumber: integer): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) do begin
    if (Result.mDest as TNodeInt).mNumber = aNumber then Break;
    Result:= OutLinkNext;
  end;
  OutPosPop;
end;

function TNodeInt.GetName: string;
begin
  Result:= IntToStr(mNumber);
  while Length(Result) < 3 do Insert('0', Result, 1);
end;

////////////////////////////////////////////////////////////////////////////
// TLink - элемент связи между вершинами
////////////////////////////////////////////////////////////////////////////

constructor TLink.Create(aOwner, aDest: TNode; aVal: integer;
                         aLoadLinks: boolean);
begin
  inherited Create;          // унаследованный конструктор
  mOwner:= aOwner;           // узел-источник связи и её владелец
  mDest:= aDest;             // узел-приёмник связи
  mLoadLinks:= aLoadLinks;   // признак нагруженной связи
  if aLoadLinks              // если нагружена
```

```
    then mValue:= aVal       // то запоминаем "вес"
    else mValue:= 1;         // а иначе по умолчанию = 1
end;

// Поиск встречного линка

function TLink.GetReverse: TLink;
begin
  Result:= mDest.GetLink(mOwner);
end;

function TLink.Copy: TItem;
begin
  Result:= TLink.Create(nil, nil, mValue, mLoadLinks);
end;


function TLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Сравниваем источники дуг
  Result:= mOwner.Compare((arg as TLink).mOwner);
  if Result in [cmpLess, cmpGreate] then Exit;
  // Сравниваем приёмники дуг
  Result:= mDest.Compare((arg as TLink).mDest);
end;

// Печать только приёмника и длины

procedure TLink.Print(var aFile: TextFile);
begin
  Write(aFile, ' '+mDest.GetName);
  if mLoadLinks then Write(aFile, '=', mValue:2);
end;

// Печать источника, приёмника и длины

procedure TLink.Print2(var aFile: TextFile);
var N1, N2 : TNode;
begin
  N1:=mOwner;  N2:=mDest;
  if not mOwner.mOwner.mDirect and
     (mOwner.Compare(mDest) = cmpGreate) then begin
    N1:=mDest;   N2:=mOwner;
  end;
  Write(aFile, ' ' + N1.GetName + '-' + N2.GetName);
  if mLoadLinks then Write(aFile, '=', mValue:3, '':1);
end;

////////////////////////////////////////////////////////////////////////////
// TGraph  -- базовый класс для графов
////////////////////////////////////////////////////////////////////////////

constructor TGraph.Create(const aName: string;
                          aDir, aLNodes, aLLinks: boolean);
begin
  inherited Create;          // вызов унаследованного конструктора
  mName:= aName;             // имя графа
  mDirect := aDir;           // признак орграфа
  mLoadNodes := aLNodes;     // признак нагруженных вершин
  mLoadLinks := aLLinks;     // признак нагруженных связей
```

```
  mNodes:= CreateSet;        // пустое множество вершин
  mMapDirect:= Undefined;    // направление для карты дальних связей
end;


destructor TGraph.Destroy;
begin
  DoneMap;                // ликвидация карты дальних указателей
  mNodes.ClrAndDestroy;   // ликвидация всех вершин
  mNodes.Free;            // ликвидация множества
  inherited;
end;

// Вспомогательные методы:

// Возвращает количество вершин

function TGraph.Nodes: integer;
begin
  Result:= mNodes.GetCount;
end;

// Выбор первой вершины

function TGraph.NodeFirst: TNode;
begin
  Result:= mNodes.GetFirst as TNode;
end;

// Выбор следующей вершины

function TGraph.NodeNext: TNode;
begin
  Result:= mNodes.GetNext as TNode;
end;

// Сохранение позиции перебора

procedure TGraph.PosPush;
begin
  mNodes.PositionPush;
end;

// Восстановление позиции перебора
procedure TGraph.PosPop;
begin
  mNodes.PositionPop;
end;

// Сброс вершин перед построением кратчайших путей

procedure TGraph.ResetNodes;
var N : TNode;
begin
  PosPush;
  N:= NodeFirst;
  while Assigned(N) do begin
    N.ResetNode;
    N:= NodeNext;
  end;
  PosPop;
end;
```

```
// Поиск копии узла в графе (nil - если не существует)

function TGraph.GetNode(aNode: TNode): TNode;
begin
  Result:= mNodes.GetObject(aNode) as TNode;
end;

// Сравнение графов

function TGraph.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result <> cmpEq then Result:= cmpGreate;
end;

// Компрессия дуг: замена длин дуг кратчайшими расстояниями
// Используется для поиска гамильтонова контура

procedure TGraph.Compress;
var  N: TNode;
     L: TLink;
    FL: TFarLink;
begin
  // Строим карту с дальними указателями:
  InitMap_Floyd;
  // Заменяем длины дуг кратчайшими путями:
  N:= NodeFirst;
  while Assigned(N) do begin
    // Перебор дальних указателей:
    FL:= N.mFarLinks.GetFirst as TFarLink;
    while Assigned(FL) do begin
      if (FL.mNodeFar <> N) then begin
        // Ищем линк из данной вершины N --> mNodeFar
        L:= N.GetLink(FL.mNodeFar);
        // Ели он существует, меняем длину на кратчайшее расстояние
        if Assigned(L) then L.mValue:= FL.mDist;
      end;
      FL:= N.mFarLinks.GetNext as TFarLink;
    end;
    N:= NodeNext;
  end;
  // Удаляем карту:
  DoneMap;
end;

// Вывод графа в файл

procedure TGraph.Print(var aFile: TextFile);
begin
  Writeln(aFile, mName);
  mNodes.Print(aFile);
end;

// Вставка вершины, возвращает FALSE при попытке вставить дубликат

function TGraph.InsertNode(aNode: TNode): boolean;
begin
  Result:= mNodes.Insert(aNode)
end;

// Удаление вершины с разрывом соответствующих связей
```

```
// Вершина удаляется, но не уничтожается!

procedure TGraph.RemoveNode(aNode: TNode);
var LS, LD: TLink;
    N: TNode;
begin
  if not mNodes.Exist(aNode) then Exit;
  // Две версии: для орграфа и неорграфа
  if mDirect then begin // ОРИЕНТИРОВАННЫЙ ГРАФ
    // Дуга образована одним линком, вставленным в два множества
    // Обработка исходящих линков удаляемой вершины:
    LS:= aNode.OutLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mDest;               // концевая вершина
      N.mLnkIn.Delete(LS);        // удаляем из множества входящих линков
      LS.Free;
      LS:= aNode.OutLinkNext;
    end;
    // Обработка входящих линков удаляемой вершины:
    LS:= aNode.InLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mOwner;              // концевая вершина
      N.mLnkOut.Delete(LS);       // удаляем из множества исходящих линков
      LS.Free;                    // уничтожаем неактуальный линк
      LS:= aNode.InLinkNext;
    end;
  end else begin // НЕОРИЕНТИРОВАННЫЙ ГРАФ
    // Ребро образовано двумя линками, вставленными в множества mLnkOut
    // Обработка исходящих линков удаляемой вершины:
    LS:= aNode.OutLinkFirst;
    while Assigned(LS) do begin
      N:= LS.mDest;               // концевая вершина
      // В концевой вершине уничтожаем встречный линк на удаляемую вершину:
      LD:= N.OutLinkFirst;
      while Assigned(LD) and (LD.mDest <> aNode)
        do LD:= N.OutLinkNext;
      N.mLnkOut.Delete(LD);       // удаляем встречный линк
      LD.Free;                    // и уничтожаем его
      LS.Free;                    // уничтожаем неактуальный прямой линк
      LS:= aNode.OutLinkNext;
    end;
  end;  // if
  // Очищаем множества связей
  with aNode do begin
    mLnkOut.Clear;
    if Assigned(mLnkIn) then mLnkIn.Clear;
  end;
  mNodes.Delete(aNode);    // удаляем вершину из графа
end;

// Установка связи дугой или ребром
// aSource - вершина-источник
// aDest   - вершина-приёмник
// aVal    - "вес" дуги или ребра

procedure TGraph.SetLink(aSource, aDest: TNode; aVal: integer);
begin
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  aSource.MakeLink(aDest, aVal);          // уст. прямой связи
  if not mDirect                          // если не орграф,
    then aDest.MakeLink(aSource, aVal);   // то уст. обратной связи
end;
```

```
// Установка степеней для всех исходящих дуг

procedure TGraph.SetLinksLimit(aLimit: integer);
var N: TNode;
begin
  N:= NodeFirst;
  while Assigned(N) do begin
    N.SetLinksLimit(aLimit);
    N:= NodeNext;
  end;
end;

//#############################################################################
//                                                                            #
//               Копирование графа и порождение подграфов                     #
//                                                                            #
//#############################################################################

// Создание копии данного графа с возможностью реверса дуг

function TGraph.CopyGr(aRevers: boolean): TGraph;
var  N1, N2 : TNode;
         ND : TNode;
          L : TLink;
begin
  Result:= TGraph.Create(mName, mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин
  N1:= NodeFirst;
  while Assigned(N1) do begin
    N2:= TNode(N1.Copy);
    Result.InsertNode(N2);
    N1:= NodeNext;
  end;
  // Копирование связей
  N1:= NodeFirst;   // первый узел исходного
  while Assigned(N1) do begin
    N2:= Result.GetNode(N1);     // узел копии
    L:= N1.OutLinkFirst;            // связь в исходном
    while Assigned(L) do begin
      ND:= Result.GetNode(L.mDest);  // приёмник в копии
      // связать узлы в зависимочти от признака реверса
      if aRevers
        then Result.SetLink(ND, N2, L.mValue)
        else Result.SetLink(N2, ND, L.mValue);
       L:= N1.OutLinkNext;  // следующая связь
    end;
    N1:= NodeNext;   // следующий узел исходного
  end;
end;

// Создание копии данного графа без реверса дуг

function TGraph.Copy: TItem;
begin
  Result:= CopyGr(false);
end;

// Порождение подграфа подмножеством его вершин

function TGraph.CopyByNodes(aNodes: TSet): TGraph;
var  N1, N2 : TNode;    // исходный узел и его копия
```

```
          ND : TNode;     // приёмник в копии
           L : TLink;
begin
  Result:= TGraph.Create('CopyByNodes', mDirect, mLoadNodes, mLoadLinks);
  // Копирование вершин:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin   // если существует в множестве аргумента
      N2:= N1.Copy as TNode;         // то создаём копию
      Result.InsertNode(N2);         // и вставляем в граф
    end;
    N1:= NodeNext;
  end;
  // Копирование связей:
  N1:= NodeFirst;                    // первый узел исходного
  while Assigned(N1) do begin
    if aNodes.Exist(N1) then begin   // если существует в множестве аргумента
      N2:= Result.GetNode(N1);       // узел копии
      L:= N1.OutLinkFirst;           // связь в исходном
      while Assigned(L) do begin     // перебор связей исходного
        ND:= Result.GetNode(L.mDest);      // приёмник в копии
        Result.SetLink(N2, ND, L.mValue); // связать узлы
        L:= N1.OutLinkNext;          // следующая связь
      end;
    end;
    N1:= NodeNext;    // следующий узел исходного
  end;
end;

// Порождение подграфа подмножеством его дуг (рёбер)

function TGraph.CopyByLinks(aLinks: TSet): TGraph;
var  NS, ND : TNode;
           L : TLink;
begin
  Result:= TGraph.Create('CopyByLinks', mDirect, mLoadNodes, mLoadLinks);

  // Порождение и связывание вершин, инцидентных заданным дугам

  L:= TLink(aLinks.GetFirst);       // Первая связь из множества
  while Assigned(L) do begin
    NS:= Result.GetNode(L.mOwner); // Взять источник связи в копии
    if not Assigned(NS)            // если нет
      then begin
        NS:= TNode(L.mOwner.Copy); // то создать
        Result.InsertNode(NS);     // и втавить
      end;
    ND:= Result.GetNode(L.mDest);  // Взять приёмник связи в копии
    if not Assigned(ND)            // если нет
      then begin
        ND:= TNode(L.mDest.Copy);  // то создать
        Result.InsertNode(ND);     // и втавить
      end;
    Result.SetLink(NS, ND, L.mValue); // связать узлы
    L:= TLink(aLinks.GetNext); // Следующая связь из множества
  end;
end;
```

```
//#########################################################################
//                                                                        #
//          Достижимость, минимальные расстояния и пути                   #
//                                                                        #
//#########################################################################

// Возвращает TRUE при наличии пути из вершины arg1 в вершину arg2
// (граф и орграф)

function TGraph.TestLink(arg1, arg2: TNode): boolean;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
    old : integer; // мощность предыдущей гаммы
      i : integer;
begin
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    Result:= SN1.Exist(arg2);   // конечная вершина в гамме?
    if Result then Break;       // да, выход из цикла
    old:= SN2.GetCount;         // запоминаем мощность
    // расширяем гамму SN2 := Gamma(SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если не расширилась
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
end;

// Возвращает количество шагов из вершины arg1 в вершину arg2
// без учёта веса рёбер или дуг
// (граф и орграф)

function TGraph.CalcSteps(arg1, arg2: TNode): integer;
var SN1 : TSet;     // предыдущее множество вершин
    SN2 : TSet;     // следующее  множество вершин (накопитель)
   node : TNode;    // очередная вершина из SN1
     ok : boolean;  // признак достижения целевой вершины
    old : integer;  // предыдущая мощность накопленной гаммы
      i : integer;
begin
  Result:= 0;
  SN1:= CreateSet;  SN2:= CreateSet;
  SN1.Insert(arg1); // нулевая гамма состоит из исходной вершины
  repeat
    ok:= SN1.Exist(arg2);  // признак достижения целевой вершины
    if ok then Break;
    old:= SN2.GetCount;    // запоминаем мощность до расширения
    // Накопление гаммы:  SN2 := Gamma (SN1)
    for i:= 1 to SN1.GetCount do begin
      node:= TNode(SN1.GetItem(i)); // node - узел исходной гаммы
      node.OutGammaAdd(SN2);        // SN2:= SN2 + Gamma(node)
    end;
    if SN2.GetCount = old then Break;  // выход, если гамма не расширилась
    Inc(Result);
    SN1.CopyItems(SN2);   // SN1 := SN2
  until false;
  SN2.Free;  SN1.Free;
```

```
  if not ok then Result:= -1;    // признак недостижимости
end;

// Генерация кратчайшего пути между двумя вершинами
// без учёта веса дуг и узлов (граф и орграф)
// aSource - исходная вершина
// aDest - целевая вершина

function TGraph.GenPath(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
     Node : TNode;       // текущая вершина
     Link : TLink;       // исходящая связь
       OK : boolean;     // признак достижения целевой вершины
begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)
  ResetNodes;               // сброс узлов в исходное состояние
  aSource.mColor:= CGray;  // исходная = серая
  Que.Put(aSource);         // исходную в очередь
  // повторяем, пока очередь не пуста:
  repeat
    OK:= false;                    // признак достижения конечной вершины
    Node:= TNode(Que.Get);         // извлечь очередную вершину
    Link:= Node.OutLinkFirst;      // первая исходящая связь
    while Assigned(Link) do begin // перебор исходящих связей
      with Link do if mDest.mColor = CWhite then begin
        // если смежная не тронута (белая)
        mDest.mPred:= Node;        // указатель на предыдущую
        OK:= mDest = aDest;        // целевая?
        if OK then Break;          // выход из цикла while
        mDest.mColor:= CGray;      // иначе красим серым
        Que.Put(mDest);            // и ставим в очередь
      end;
      Link:= Node.OutLinkNext; // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;      // обработанную вершину красим чёрным
    // выход, если достигнута целевая либо исчерпана очередь
  until OK or (Que.GetCount = 0);

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Put(aDest);                   // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Put(Node);                  // вершину --> в результат
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;       // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

```
// Генерация кратчайшего пути между двумя вершинами
// по алгоритму Дейкстры с учётом весов рёбер и вершин
// (граф и орграф)

function TGraph.GenDijkstra(aSource, aDest: TNode): TBuffer;

var  Que : TBuffer;  // очередь вершин
     Node : TNode;      // текущая вершина
     Link : TLink;      // исходящая связь

begin
  Result:= nil;
  if aSource = aDest then Exit;  // если вершины совпадают
  Que:= TBuffer.Create;  // создаём рабочий буфер (очередь)

  ResetNodes;               // сброс узлов в исходное состояние
  aSource.mColor:= CGray;   // исходная = серая
  aSource.mDist:= 0;        // расстояние к исходной = 0
  Que.Put(aSource);         // исходную в очередь

  repeat                    // повторяем, пока очередь не пуста
    Node:= TNode(Que.Get); // извлечь очередную вершину
    Link:= Node.OutLinkFirst; // первая исходящая связь
    while Assigned(Link) do with Link do begin  // обраб. текущего линка
      if mDest.mDist > Node.mDist + Ord(mLoadNodes)*Node.mValue + mValue
        then begin
        // поскольку расстояние в смежной больше нового,
        // обновляем расстояние и обратную ссылку
        mDest.mDist:= Node.mDist + Ord(mLoadNodes)*Node.mValue + mValue;
        mDest.mPred:= Node;
        // Если смежная не стоит в очереди, то красим белым,
        // чтобы поставить её в очередь (возможно, что повторно)
        if mDest.mColor <> CGray then mDest.mColor:= CWhite;
      end;
      if mDest.mColor = CWhite then begin
        // если смежная не стояла в очереди
        mDest.mColor:= CGray;   // то красим серым
        Que.Put(mDest);         // и ставим в очередь
      end;
      Link:= Node.OutLinkNext;  // следующая исходящая связь
    end; // while
    Node.mColor:= CBlack;   // признак, что вершина обработана
  until Que.GetCount = 0;   // выход, если исчерпана очередь

  // Формирование результата в буфере Que
  if Assigned(aDest.mPred) then begin // если достигнута целевая
    Que.Clear;                        // очистить буфер
    Que.Put(aDest);                   // и вставить целевую вершину
    Node:= aDest.mPred;               // начало перебора обратных ссылок
    while Assigned(Node) do begin     // пока существуют
      Que.Put(Node);                  // вершину --> в результат
      if Node = aSource then Break;   // выход, если достигли исходную
      Node:= Node.mPred;              // следующая обратная ссылка
    end;
    Que.Reversion;                    // реверсируем буфер
    Result:= Que;
  end else begin
    Que.Free;   // если целевая не достигнута, ликвидируем буфер
  end;
end;
```

```
// Генерация карты маршрутизации по алгоритму Флойда-Уоршелла
// для быстрого поиска кратчайших путей
// (граф и орграф)
// Майника, "Алгоритмы оптимизации на сетях и графах", стр. 53

// TFarLink - объект для карты маршрутов

constructor TFarLink.Create(aNear, aFar: TNode; aDist: integer);
begin
  inherited Create;
  mNodeNear:= aNear;  // соседняя вершина
  mNodeFar:= aFar;    // целевая вершина
  mDist:= aDist;      // расстояние к целевой вершине
end;

// Упорядочено в порядке неубывания дистанции

function TFarLink.Compare(arg: TItem): TCompare;
begin
  if mDist > (arg as TFarLink).mDist
    then Result:= cmpGreate
  else if mDist < (arg as TFarLink).mDist
    then Result:= cmpLess
  else Result:= mNodeFar.Compare((arg as TFarLink).mNodeFar)
end;


procedure TFarLink.Print(var aFile: TextFile);
begin
  Write(aFile,'Far  = ');          // целевая вершина
  if Assigned(mNodeFar)
    then mNodeFar.Print(aFile)
    else Writeln('NIL');
  Write(aFile,'Near = ');          // соседняя вершина
  if Assigned(mNodeNear)
    then mNodeNear.Print(aFile)
    else Writeln('NIL');           // расстояние к целевой вершине
  Writeln(aFile,'Dist = ', mDist);
  Writeln(aFile,'- - - - - - -');
end;

// Метод генерации карты маршрутизации по алгоритму Флойда-Уоршелла

procedure TGraph.InitMap_Floyd;

  //- - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
  begin
    PosPush;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // связь на себя
      end else with aNode do begin
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        Link:= OutLinkFirst;           // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = Node then begin
```

```
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue; // + Ord(mLoadNodes)*Node.mValue;
            FL.mStep:= 1;      // указатель будет обработан на первом шаге
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end; // while
      end; // else
      aNode.mFarLinks.Insert(FL);        // вставить в множество
      Node:= NodeNext;
    end;
    PosPop;
  end;
  //- - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
          FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;  // указатель из вершины
aNode
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
              then Result:= true;           // признак того, что изменён
          FL2:= GetNext as TFarLink;        // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - -
```

```
var  Node : TNode;     // текущая вершина
     Step : integer;   // этап обработки (номер цикла)
     Flag : boolean;   // признак продолжения обработки

begin { InitMap_Floyd }

  // Предварительная очистка карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin         // перебор вершин
    with Node do begin
      if Assigned(mFarLinks)            // если карта существует
        then mFarLinks.ClrAndDestroy    // очищаем
        else mFarLinks:= CreateSet;     // иначе создаём пустую
    end;
    InitFarLinks(Node);    // инициализация дальних указателей
    Node:= NodeNext;
  end;

  // Обработка вершин
  for Step:= 1 to mNodes.GetCount-2  do begin
    Flag:= false;
    Node:= NodeFirst;
    while Assigned(Node) do begin      // перебор вершин
      if Handle(Node, Step)            // если обновлялись дальние указатели
        then Flag:= true;              // то отметить
      Node:= NodeNext;
    end;
    if not Flag then Break;  // если указатели не обновлялись, то выйти
  end;
end;

// Поиск кратчайшего маршрута по карте маршрутизации
// с формированием буфера

function TGraph.GenQuickPath(aSource, aDest: TNode): TBuffer;
var Next: TNode;

begin
  Result:= nil;
  // Если карта ещё не построена, то строим её
  if not mInitMap then InitMap_Floyd;
  mInitMap:= true; // Уст. признак инициализации карты маршрутизации

  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;

  Result:= TBuffer.Create;   // созд. буфер результата
  Result.Put(aSource);         // заносим начальную вершину
  // Следуем по маршруту:
  while Next <> aDest do begin
    Result.Put(Next);          // заносим промежуточную вершину
    Next:= Next.GetNear(aDest);
  end;
  Result.Put(Next);    // заносим конечную вершину
end;
```

```
// Поиск кратчайшего маршрута по карте маршрутизации
// с формированием строки

function TGraph.GetQuickPathStr(aSource, aDest: TNode): String;
var Next : TNode;
      FL : TFarLink;
       S : String;
begin
  Result:= aSource.GetName;
  // Если карта ещё не построена, то строим её
  if not mInitMap then InitMap_Floyd;
  mInitMap:= true; // Уст. признак инициализации карты маршрутизации
  // Проверяем достижимость целевой вершины:
  Next:= aSource.GetNear(aDest);
  if not Assigned(Next) then Exit;   // если не достижима
  // Следуем по маршруту:
  repeat
    Result:= Result + ' -> ' + Next.GetName;
    Next:= Next.GetNear(aDest);
  until Next = aDest;
  FL:= aSource.GetFarLink(aDest); // дальний указатель на конечную
  Str(FL.mDist, S);               // растояние до конечной
  Result:= Result + ' -> ' + Next.GetName + ' : ' + S;
end;

//#############################################################################
//                                                                            #
//                    Области связности                                       #
//               Кристофидес, Глава 2 стр. 29                                 #
//                                                                            #
//#############################################################################

// TCondensNode --  конденсат
// Конструктор принимает сильную компонету графа
// (множество сильно связанных вершин)

constructor TCondensNode.Create(aNodes: TSet; aOwner: TGraph);
var Name : string;
     Val : integer;
      N  : TNode;
begin
  Name:='';  Val:= 0;
  // Цикл формирования имени и цены (веса)
  N:= TNode(aNodes.GetFirst);
  while Assigned(N) do begin
    Name:= Name+ N.GetName;       // объединение имён
    Val:= Val + N.mValue;         // и весов вершин
    N:= TNode(aNodes.GetNext);
  end;
  inherited Create(Val, aOwner);
  mName:= Name;
  // Копируем сильную компоненту
  mNodes:= TSet(aNodes.Copy);
  // Формируем mOut - ссылки за пределы сильной компоненты (узлы)
  mOut:= CreateSet;
  N:= TNode(mNodes.GetFirst);
  while Assigned(N) do begin
    N.OutGammaAdd(mOut);
    N:= TNode(mNodes.GetNext);
  end;
  mOut.Sub(mNodes);  // за исключением внутренних узлов СК
end;
```

```
destructor TCondensNode.Destroy;
begin
  mNodes.Free;
  mOut.Free;
  inherited;
end;

function TCondensNode.GetName: string;
begin
  Result:= mName;
end;

function TCondensNode.Copy: TItem;
begin
  Result:= nil;
end;


// Проверка связности (граф и орграф)

function TGraph.IsLinked: boolean;
var Node: TNode;
    S : TSet;
    NIn, NOut : integer;
begin
  Result:= false;
  // Берём произвольную вершину:
  Node:= NodeFirst;
  if not Assigned(Node) then Exit;
  // Подсчитываем вершины, из которых достижима вершина Node:
  S:= Node.GenGammaIn;
  NIn:= S.GetCount;
  S.Free;
  // Подсчитываем вершины, которые достижимы из вершины Node:
  S:= Node.GenGammaOut;
  NOut:= S.GetCount;
  S.Free;
  // Граф связан или сильно связан, если оба множества
  // покрывают все вершины графа:
  Result:= (mNodes.GetCount = NIn) and (mNodes.GetCount = NOut);
end;

// Подсчёт количества (сильно) связных областей
function TGraph.CalcAreas: integer;
var Copy: TSet;    // копия множества всех вершин графа
    Node: TNode;   // текущая вершина
    Strong: TSet;  // область сильной связи текущей вершины
begin
  Result:= 0;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Inc(Result);                      // счётчик областей
    Node:= Copy.GetFirst as TNode;    // первая вершина из копии
    Strong:= Node.GenStrongArea;      // область её связности
    Copy.Sub(Strong);                 // вычитаем из копии
    Strong.Free                       // освобождаем область
  until Copy.GetCount = 0;
  Copy.Free;
end;
```

```
// Генерация множества (сильно) связных областей (подграфов)

function TGraph.GenAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    Node: TNode;    // текущая вершина
    Strong: TSet;   // область сильной связи текущей вершины
    Gr: TGraph;     // граф, порождённый на множестве вершин Strong
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Обработка копии:
  repeat
    Node:= Copy.GetFirst as TNode;  // первая вершина из копии
    Strong:= Node.GenStrongArea;    // область её связности
    Gr:= CopyByNodes(Strong);       // создание порождённого подграфа
    Result.Insert(Gr);              // вставка порождённого подграфа
    Copy.Sub(Strong);               // вычитаем из копии
    Strong.Free                     // освобождаем область
  until Copy.GetCount = 0;          // пока копия не пуста
  Copy.Free;
end;

// Формирование множества-конденсата
// (подмножеств сильных компонент)
// Кристофидес, стр. 33

function TGraph.GenStrongAreas: TSet;
var Copy: TSet;     // копия множества всех вершин графа
    N : TNode;
    Strong: TSet;  // сильная компонента
begin
  Result:= CreateSet;
  // Создаём множество-копию всех вершин
  Copy:= CreateSet;  Copy.CopyItems(mNodes);
  // Пока копия не пуста:
  while Copy.GetCount <> 0 do begin
    N:= Copy.GetFirst as TNode;       // первая вершина в копии
    Strong:= N.GenStrongArea;         // находим сильную компоненту
    Result.Insert(Strong);            // и вставляем в результат
    Copy.Sub(Strong);                 // удаляем её из копии
  end;
  Copy.Free;  // освобождаем копию
end;

// Создание графа-конденсата
// Кристофидес, стр. 33

function TGraph.GenCondens: TGraph;
var Strongs : TSet;          // множество сильных компонент
          S : TSet;          // вспомогательное множество
         CN : TCondensNode; // текущий узел-конденсат
     Target : TCondensNode; // целевой узел-конденсат
begin
  Result:= TGraph.Create('Condens', true, false, false);  // пустой граф
  Strongs:= GenStrongAreas;                // формируем множество-конденсат
  S:= Strongs.GetFirst as TSet;            // подмнож. вершин в узле
конденсата
  while Assigned(S) do begin               // перебор множеств конденсата
    CN:= TCondensNode.Create(S, Result);  // создаём вершину конденсата
    Result.InsertNode(CN);                 // и вставляем в граф-конденсат
```

```
    S:= Strongs.GetNext as TSet;          // подмнож. вершин в узле
конденсата
  end;
  // формирование связей между вершинами конденсата:
  S:= CreateSet;  // вспомогат. множество вершин исходного графа
  CN:= Result.NodeFirst as TCondensNode;        // перебор вершин конденсата
  while Assigned(CN) do begin
    if CN.mOut.GetCount <> 0 then begin         // если степень исхода <> 0
      Result.PosPush;                           // сохр. позицию перебора
      Target:= Result.NodeFirst as TCondensNode;// вершина конденсата
      while Assigned(Target) do begin           // внутр. цикл перебора
        if CN <> Target then begin              // если не текущая вершина
          S.CopyItems(CN.mOut);                 // исходящие связи текущей
          S.Mul(Target.mNodes);                 // пересекаем с целевыми
          if S.GetCount<>0                      // если есть связи в исх.
графе
            then Result.SetLink(CN, Target, 1); // то уст. связь в конденсате
        end;
        Target:= Result.NodeNext as TCondensNode; // след. узел конденсата
      end;
      Result.PosPop;                           // восст. позицию перебора
    end;
    CN:= Result.NodeNext as TCondensNode;      // след. узел конденсата
  end;
  S.Free;
  Strongs.ClrAndDestroy;  // очищаем множество-конденсат
  Strongs.Free;
end;

//#########################################################################
//                                                                        #
//     Формирование максимальных подмножеств независимых вершин           #
//                    Кристофидес-2.3,  стр. 46                           #
//               Алгоритм_Брона_—_Кербоша (Bron-Kerbosh)                  #
//                     Неориентированый граф                              #
//            Результат - множество независимых подмножеств               #
//                                                                        #
//#########################################################################

function TGraph.GenUndepend: TSet;
var
  Res: TSet;    // очередное максимальное независимое множество
  Gamma: TSet;  // для доступа к соседним вершинам
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает TRUE, если среди обработанных вершин (aTested)
  // есть хоть одна, не соседствующая ни с одним кандидатом (aCand)

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);   // первый из обработанных
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи обработанного
      Gamma.Mul(aCand);        // Gamma = пересечение соседей с кандидатами
      if Gamma.GetCount=0 then begin // если пересечение пусто,
        Result:= true;       // значит не соседствует ни с одним кандидатом
        Break;
      end;
      Node:= TNode(aTested.GetNext);  // следующий из обработанных
    end;
  end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура расширения независимого множества
  // aCand, aTested - множества с предыдущего уровня

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // новые вершины-кандидаты
    newTested: TSet; // новые обработанные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и
    // нет независимых от них обработанных вершин
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst);     // взять первого кандидата
      Res.Insert(Node);                 // вставить в результат
      Node.OutGammaGet(Gamma);          // Gamma= его соседи
      // Создать копии множеств кандидатов и проверенных
      newCand:= TSet(aCand.Copy);
      newTested:= TSet(aTested.Copy);

      newCand.Delete(Node);   // удалить текущий узел из кандидатов
      newCand.Sub(Gamma);     // и удалить его соседей
      newTested.Sub(Gamma);   // удалить соседей из копии проверенных

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
             // то найдено очередное макс. независимое множество
             Result.Insert(Res.Copy)
        else if newCand.GetCount<>0 then
               // если остались кандидаты, то рекурсивный вызов
               // с новыми наборами кандидатов и проверенных
               Extend(newCand, newTested);

      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
      Res.Delete(Node);     // удалить текущий из результата
      aCand.Delete(Node);   // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Cand : TSet;   // вершины-кандидаты
  Tested: TSet;  // проверенные вершины

begin  { TGraph.GenUndepend}

  // Инициализация
  Result:= CreateSet;       // создать пустое множество-результат
  Res:= CreateSet;          // очередное макс. независимое множество
  Cand := CreateSet;        // кандидаты
  Cand.CopyItems(mNodes);   // Cand = все вершины графа
  Tested:= CreateSet;       // проверенные вершины
  Gamma:= CreateSet;        // соседние вершины

  Extend(Cand, Tested); // рекурсив. проц. построения независ. множеств
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
```

```
  Res.Free;
end;

//#########################################################################
//                                                                        #
//            Формирование множества максимальных  клик                   #
//                  Кристофидес-2.3,  стр. 46                             #
//               Алгоритм_Брона_—_Кербоша ( Bron-Kerbos                   #
//                     Неориентированый граф                              #
//                                                                        #
//#########################################################################

function TGraph.GenClique: TSet;
var
  Res: TSet;      // очередная клика
  Gamma : TSet;  // соседние вершины
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает TRUE, если среди проверенных вершин в aTested
  // есть хоть одна, примыкающая ко всем кандидатами в aCand

  function  Test(aCand, aTested: TSet): boolean;
  var Node: TNode;
  begin
    Result:= false;
    Node:= TNode(aTested.GetFirst);
    while Assigned(Node) do begin
      Node.OutGammaGet(Gamma); // Gamma = соседи Node
      Gamma.Mul(aCand);        // Gamma = соседи * кандидаты
      if Gamma.GetCount = aCand.GetCount then begin
        // если примыкает ко всем кандидатам
        Result:= true;
        Break;
      end;
      Node:= TNode(aTested.GetNext);
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура расширения клики

  procedure Extend(aCand, aTested : TSet);
  var
    newCand : TSet;  // вершины-кандидаты
    newTested: TSet; // проверенные вершины
    Node: TNode;     // очередная вершина
  begin
    // Пока есть кандидаты и в проверенных aTested нет таких,
    // что примыкают ко всем кандидатами в aCand
    while (aCand.GetCount<>0) and not Test(aCand, aTested) do begin

      Node:= TNode(aCand.GetFirst); // очередной кандидат
      Res.Insert(Node);             // вставляем в клику
      Node.OutGammaGet(Gamma);      // Gamma:= соседи Node

      newCand:= TSet(aCand.Copy);   // копия кандидатов
      newCand.Delete(Node);         // удаляем текущую вершину
      newCand.Mul(Gamma);           // и оставляем только соседние

      newTested:= TSet(aTested.Copy); // копия проверенных
      newTested.Mul(Gamma);           // оставляем только соседние

      if (newCand.GetCount=0) and (newTested.GetCount=0)
        then // если не осталось кандидатов и ничего не пропущено,
```

```
            // то найдена очередная клика
            Result.Insert(Res.Copy)

        else if newCand.GetCount<>0 then
            // если остались кандидаты, то рекурсивный вызов
            // с новыми наборами кандидатов и проверенных
            Extend(newCand, newTested);

      // Возвращение
      newTested.Free;       // удалить копию проверенных
      newCand.Free;         // удалить копию кандидатов
      Res.Delete(Node);     // удалить текущий из результата
      aCand.Delete(Node);   // удалить текущий из кандидатов
      aTested.Insert(Node); // вставить текущий в проверенные
    end
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Cand : TSet;   // вершины-кандидаты
  Tested: TSet;  // проверенные вершины

begin   { TGraph.GenClique }

  // Инициализация
  Result:= CreateSet;      // создать пустое множество-результат
  Res:= CreateSet;         // очередная клика
  Cand := CreateSet;       // кандидаты
  Cand.CopyItems(mNodes);  // Cand = все вершины графа
  Tested:= CreateSet;      // проверенные вершины
  Gamma:= CreateSet;       // соседние вершины

  Extend(Cand, Tested);  // рекурсивная процедура
  // Освобождение памяти
  Gamma.Free;
  Tested.Free;
  Cand.Free;
  Res.Free;

end;

//#########################################################################
//                                                                        #
//           Формирование доминирующего множества                        #
//                 Кристофидес-3,  стр. 50                                #
//                      Граф + орграф                                     #
//                                                                        #
//#########################################################################

// TCostNodes - Элемент для ппоиска доминирующих множеств

constructor TCostNodes.Create(aCost: integer; aSet: TSet; aNode: TNode);
begin
  inherited Create(aCost, aSet, true);
  mNode:= aNode;
end;


procedure TCostNodes.Print(var aFile: TextFile);
var Node: TNode;
begin
  Write(aFile, mNode.GetName, ' = ', mCost, ' -> { ');
  Node:= TNode(mSet.GetFirst);
```

```
  while Assigned(Node) do begin
    Write(aFile, Node.GetName + ' ');
    Node:= TNode(mSet.GetNext);
  end;
  Writeln(aFile, '}');
end;

// Метод поиска доминирующих множеств

function TGraph.GenDominating: TCostSet;
var  Buf: TBuffer;    // промежуточный буфер
     Node: TNode;       // Вершина графа
     Gamma: TSet;       // для множества соседей
     CS: TCostNodes;    // оценённое подмножество

begin
  Buf:= TBuffer.Create;
  // В промежуточный буфер заносим оценённые подмножества,
  // каждое из которых соответствует гамме одной вершины
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Gamma:= CreateSet;         // создаём пустое множество
    Node.OutGammaGet(Gamma);   // Gamma= соседние вершины
    Gamma.Insert(Node);        // + текущая
    // Node.CalcCost вычисляет цену узла := вес узла + веса дуг
    CS:= TCostNodes.Create(Node.CalcCost, Gamma, Node);
    Buf.Put(CS);          // заносим в буфер
    Node:= NodeNext;
  end;
  // Поиск минимального покрытия
  Result:= CollectMinCover(mNodes, Buf);  // unit Assembly
  // Удаляем результат из буфера {unit SetUtils}
  RemoveItemsFromBuf(Buf, Result.mSet);
  // а буфер очищаем и уничтожаем
  Buf.ClrAndDestroy;
  Buf.Free;
end;

//#############################################################################
//                                                                            #
//                            Раскраска                                       #
//                    Кристофидес-4,  стр. 86                                 #
//                                                                            #
//#############################################################################

// Построение раскраски
// aQuick - параметр скорости:
// false = точно, но долго
// true  = быстро, но неточно

function TGraph.GenPaints(aQuick: boolean): TSet;
var  Undep : TSet;        // множество независимых множеств
       Buf : TBuffer;  // рабочий буфер
         S : TSet;        // текущее подмножество вершин
        CS : TCostSet;    // оценённое подмножество вершин
     Cover : TCostSet;    // минимальное покрытие
begin
  Undep:= GenUndepend;    // генерируем все независимые подмножества вершин
  // Переносим независимые множества в буфер оценённых подмножеств
  // для последующего поиска минимального покрытия
  Buf:= TBuffer.Create;       // создаём рабочий буфер
  S:= Undep.GetFirst as TSet;  // первое независимое множество
```

```
  while Assigned(S) do begin   // перебор независимых множеств
    // Создаём оценённые подмножества (цена Cost=1)
    // и помещаем в буфер:
    CS:= TCostSet.Create(1, S.Copy as TSet, true);
    Buf.Put(CS);
    S:= Undep.GetNext as TSet;
  end;
  // Уничтожаем множество максимальных независимых множеств
  Undep.ClrAndDestroy;
  Undep.Free;
  // Формируем минимальное покрытие вершин (unit Assembly):
  if aQuick
    then Cover:= CollectGradCover(mNodes, Buf)  // быстро
    else Cover:= CollectMinCover(mNodes, Buf);  // точно
  RemoveItemsFromBuf(Buf, Cover.mSet);  // удаляем результат из буфера
  Buf.ClrAndDestroy;                    // а буфер очищаем
  Buf.Free;                             // и удаляем
  // Формируем результат:
  Result:= Cover.mSet.Copy as TSet;  // копия множества-покрытия
  Cover.Free;                        // удаляем покрытие
  Result.CoverToDissect;             // превращаем покрытие в разбиение
end;

//#########################################################################
//                                                                        #
//                        Центры и P-центры                               #
//                      Кристофидес-5,  стр. 111                          #
//                                                                        #
//#########################################################################

// Элемент для представления ограниченных входящих гамм
// Используется для поиска p-центров

type  TGamma = class(TItem)
        mRoot  : TNode;  // достигаемая вершина
        mGamma : TSet;   // вершины, из которых достижима mRoot
        constructor Create(aRoot: TNode; aGamma: TSet);
        destructor Destroy;  override;
        function  Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TGamma.Create(aRoot: TNode; aGamma: TSet);
begin
  inherited Create;
  mRoot:= aRoot;       // центр гаммы
  mGamma:= aGamma;     // подмножество достижимых вершин
end;

destructor TGamma.Destroy;
begin
  mGamma.Free;
  inherited;
end;

function TGamma.Compare(arg: TItem): TCompare;
begin
  Result:= mRoot.Compare((arg as TGamma).mRoot)
end;

procedure TGamma.Print(var aFile: TextFile);
var S: string;
```

```
begin
  S:= mRoot.GetName + ' -> ' + NodesToStr(mGamma);
  Writeln(aFile, S);
end;


// TSortedNum - вспомогательный класс для формирования
// множества возрастающих чисел

type  TSortedNum = class(TItem)
        mNumber : integer;
        constructor Create(aNumber : integer);
        function  Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TSortedNum.Create(aNumber: integer);
begin
  inherited Create;
  mNumber:= aNumber;
end;

function TSortedNum.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result= cmpEq then Exit;
  if mNumber < (arg as TSortedNum).mNumber
    then Result:= cmpLess
  else if mNumber > (arg as TSortedNum).mNumber
    then Result:= cmpGreate  else Result:= cmpEq  ;
end;

procedure TSortedNum.Print(var aFile: TextFile);
begin
  Writeln(aFile,'Num= ',mNumber);
end;


// TAreal
// Элемент для представления областей
// Используется для поиска p-центров
// aAreal - достигаемые вершины (поле mSet)
// aGamma - вершины, из которых они достигаются (поле mGamma)

type  TAreal = class(TCostSet)
        // поле mSet - достигаемые вершины, унаследовано от TCostSet
        mGamma : TSet;  // вершины, из которых достижимы вершины в mSet
        constructor Create(aAreal, aGamma: TSet);
        destructor Destroy;  override;
        procedure Print(var aFile: TextFile); override;
      end;

constructor TAreal.Create(aAreal {куда},
                          aGamma {откуда} : TSet);
begin
  inherited Create(1, aAreal, true);
  mGamma:= aGamma;  // вершины, из которых достижимы вершины в mSet
end;

destructor TAreal.Destroy;
begin
  mGamma.Free;
```

```
  inherited;
end;

procedure TAreal.Print(var aFile: TextFile);
var S: string;
begin
  S:= NodesToStr(mGamma) + ' ==> ' + NodesToStr(mSet);
  Writeln(aFile, S);
end;


// Функция формирования центра графа.
// Возвращает множество вершин центра и взвешенный радиус aLambda
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenCenter(var aLambda: integer; aDirect: TCenter): TSet;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск максимального взвешенного расстояния от вершины aNode.
  // Использует ранее построенную карту дальних указателей mFarLinks

  function GetMaxDist(aNode: TNode): integer;
  var FL: TFarLink;
      Value: integer;
  begin
    Result:= 0;
    with aNode.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if mLoadNodes                          // если вершины графа
нагружены...
          then Value:= FL.mNodeFar.mValue      // для графа с нагруж. вершинами
          else Value:= 1;                      // если вершины не нагружены
        if Result < FL.mDist * Value
          then Result:= FL.mDist * Value;
        FL:= GetNext as TFarLink;
      end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Dist, MinDist: integer;  // текущее и минимальное взвеш. расстояние
    Node: TNode;

begin  { TGraph.GenCenter }

  // Предварительно формируем карту дальних указателей,
  // в которой каждый элемент содержит расстояние между парами вершин.
  DoneMap;           // если карта создана, то ликвидируем её

  InitMap(aDirect);  // строим карту с заданным направлением

  // Выбор минимального из максимальных взвешенных расстояний
  MinDist:= MaxInt;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);
    if MinDist > Dist then MinDist:= Dist;
    Node:= NodeNext;
  end;
  Result:= CreateSet;
```

```
  // Формирование множества-центра
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Dist:= GetMaxDist(Node);    // число разделения очередной вершины
    if Dist = MinDist then Result.Insert(Node);
    Node:= NodeNext;
  end;
  aLambda:= MinDist;  // минимальное из максимальных расстояний
end;


// Вспомогательная процедура генерации распределённой карты
// aDirect = InCenter | OutCenter | InOutCenter

procedure TGraph.InitMap(aDirect: TCenter);

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Начальная инициализация дальних связей узла
  procedure InitFarLinks(aNode: TNode; aDirect : TCenter);
  var     Node : TNode;     // текущая вершина
          Link : TLink;     // ближний указатель
            FL : TFarLink;  // дальний указатель
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внутреннего центра орграфа или центра графа
    procedure LocalIn;
    begin
      with Node do begin
        Link:= OutLinkFirst;            // первая исходящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mDest = aNode then begin  // сравнение с конечной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= OutLinkNext;     // следующая исходящая связь
        end // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Для поиска внешнего центра орграфа
    procedure LocalOut;
    begin
      with Node do begin
        Link:= InLinkFirst;             // первая входящая связь
        while Assigned(Link) do begin   // пока существуют связи
          if Link.mOwner = aNode then begin // сравнение с исходной вершиной
            // Устанавливаем связь с ближайшей вершиной
            FL.mNodeNear:= Node;
            FL.mDist:= Link.mValue;
            FL.mStep:= 1;   // указатель будет обработан на первом этапе
            Break;
          end;
          Link:= InLinkNext;     // следующая исходящая связь
        end; // while
      end // with
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  begin   { InitFarLinks }
    PosPush;  // сохранить позицию перебора вершин
    Node:= NodeFirst;                // перебор всех вершин графа
```

```
    while Assigned(Node) do begin    // цикл создания дальних связей
      if Node = aNode then begin
        FL:= TFarLink.Create(Node, Node, 0);   // это связь на себя
      end else with Node do begin
        // Создать связь на очередную вершину
        FL:= TFarLink.Create(nil {near}, Node {far}, CInfinity);
        if (aDirect=InCenter) or not mDirect
          then LocalIn   // Внутренний центр орграфа или центр графа
          else LocalOut; // Внешний центр орграфа
      end; // else
      aNode.mFarLinks.Insert(FL); // вставить в множество
      Node:= NodeNext;  // перебор всех вершин графа
    end;
    PosPop;  // восстановить позицию перебора вершин
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка дальних связей узла (обработка строки матрицы)

  function  Handle(aNode: TNode; aStep: integer): boolean;
  var   FL1 : TFarLink;  // указатель от исходной к промежуточной вершине
        FL2 : TFarLink;  // указатель от промежуточной вершины к конечной

    // Локальная функция проверки очередной связи FL2

    function Test_FL2(aNear : TNode): boolean;
    var  Dist : integer;  // новая дистанция
          FL : TFarLink;   // указатель в исходной вершине aNode
    begin
      Result:= false;
      Dist:= FL1.mDist + FL2.mDist;  // новое расстояние через промежуточную
      FL:= aNode.GetFarLink(FL2.mNodeFar); // найти указатель на конечную
      if FL.mDist > Dist then begin    // если существующее больше нового
        FL.mNodeNear:= FL1.mNodeNear;  // то меняем путь
        FL.mDist:= Dist;               // и расстояние
        FL.mStep:= aStep+1;  // эта связь будет обработана на следующем шаге
        Result:= true;       // признак изменения дальней связи
      end;
    end;

  begin  { Handle }
    Result:= false;
    FL1:= aNode.mFarLinks.GetFirst as TFarLink;  // указатель из aNode
    while Assigned(FL1) do begin
      // Через этот линк просматриваем дальних соседей
      if FL1.mStep=aStep then with FL1.mNodeFar.mFarLinks do begin
        PositionPush;
        FL2:= GetFirst as TFarLink;   // указатель на промежуточную вершину
        while Assigned(FL2) do begin
          with FL2 do  // очередной дальний указатель в промежуточной вершине
          if Assigned(mNodeNear)             // если определён
             and (mDist<>0)                  // и не сам на себя
             and (mNodeFar<>aNode)           // и не на исходную вершину
            then if Test_FL2(FL2.mNodeNear) // то проверяем расстояние
                   then Result:= true;      // признак того, что изменён
          FL2:= GetNext as TFarLink;        // следующая дальняя связь
        end; // while
        PositionPop;
      end;
      FL1:= aNode.mFarLinks.GetNext as TFarLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  procedure Create(aDirect : TCenter);
  var  Node : TNode;     // текущая вершина
       Step : integer;   // этап обработки (номер цикла)
       Flag : boolean;   // признак продолжения обработки
  begin
    // Предварительная очистка карты:
    Node:= NodeFirst;
    while Assigned(Node) do begin          // перебор вершин
      with Node do if Assigned(mFarLinks) // если карта существует
        then mFarLinks.ClrAndDestroy       // очищаем
        else mFarLinks:= CreateSet;        // иначе создаём пустую
      InitFarLinks(Node, aDirect);         // инициализация дальних указателей
      Node:= NodeNext;
    end;
    // Обработка вершин
    for Step:= 1 to mNodes.GetCount-2  do begin
      Flag:= false;
      Node:= NodeFirst;
      while Assigned(Node) do begin // перебор вершин
        if Handle(Node, Step)       // если обновлялись дальние указатели,
          then Flag:= true;         // то отметить это установкой флага
        Node:= NodeNext;
      end;
      // если дальние указатели не обновлялись, то прервать цикл
      if not Flag then Break;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var  Node : TNode;      // текущая вершина
     Buf : TBuffer;  // буфер для хранения списков дальних указателей

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Сохранение в буфере текущих расстояний
  procedure SaveInBuf;
  begin
    Node:= NodeFirst;
    // Сохранение списков в буфере:
    while Assigned(Node) do begin
      Buf.Put(Node.mFarLinks);
      Node.mFarLinks:= nil;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение из буфера и добавление расстояний
  procedure AddFromBuf;
  var     S : TSet;       // список из буфера
        FL1 : TFarLink;   // дальний указатель из S
        FL2 : TFarLink;   // дальний указатель из Node
  begin
    Node:= NodeFirst;
    // Перебор вершин:
    while Assigned(Node) do begin
      S:= Buf.Get as TSet;
      FL1:= S.GetFirst as TFarLink;
      while Assigned(FL1) do begin
        FL2:= Node.GetFarLink(FL1.mNodeFar);
        if Assigned(FL2) then FL2.mDist:= FL2.mDist + FL1.mDist;
        FL1:= S.GetNext as TFarLink;
      end;
      S.ClrAndDestroy;
```

```
      S.Free;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.InitMap }

  // Выход, если карта уже создана
  if mMapDirect = aDirect  then Exit;

  mMapDirect:= aDirect; // запоминаем направление карты

  case aDirect of
    InCenter,   // внешний центр и медиана
    OutCenter:  // внутренний центр и медиана
      Create(aDirect);
    InOutCenter: // внешне-внутренний центр и медиана
      begin
        // Строим карту для входящих гамм:
        Create(InCenter);
        // Создаём буфер и сохраняем в нём расстояния между вершинами:
        Buf:= TBuffer.Create;
        SaveInBuf;
        // Строим карту для исходящих гамм:
        Create(OutCenter);
        // Прибавляем расстояния, сохранённые в буфере,
        // и освобождаем буфер:
        AddFromBuf;
        Buf.ClrAndDestroy;
        Buf.Free;
      end;
  end;

  // Сортируем списки дальних указателей по неубыванию расстояния
  // (необходимо при поиске медиан)

  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.SortFarLinks;
    Node:= NodeNext;
  end;

end;

// Освобождение элементов, применяемых при построении
// кратчайших путей и центров

procedure TGraph.DoneMap;
var  Node : TNode;    // текущая вершина
begin
  // Если очищена, то выход:
  if mMapDirect = Undefined then Exit;
  // Очистка распределённой по вершинам карты:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      if Assigned(mFarLinks) then begin
        mFarLinks.ClrAndDestroy;   // уничтожаем указатели
        mFarLinks.Free;            // и список
        mFarLinks:= nil;
      end;
```

```
    end;
    Node:= NodeNext;
  end;
  mMapDirect:= Undefined;  // признак очищенной карты
end;

// Отображение распределённой карты

procedure TGraph.ExpoFarLinks;
var Node: TNode;
begin
  PosPush;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.ExpoFarLinks;
    Node:= NodeNext;
  end;
  PosPop;
end;

// Отображение дополнительных данных в связях

procedure TGraph.ExpoLinksData;
var Node: TNode;
begin
  PosPush;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.ExpoData;
    Node:= NodeNext;
  end;
  PosPop;
end;

// Формирование множества вершин, из которых узел aNode
// достижим в пределах взвешенного расстояния aLambda
// (ограниченная входящая гамма)
// Используется созданная ранее карта дальних указателей.

function TGraph.GenLimitGamma(aNode: TNode; aLambda: integer): TSet;
var FL : TFarLink;    // очередная дальняя связь
begin
  Result:= CreateSet;
  with aNode.mFarLinks do begin  // просматриваем дальние связи вершины
    FL:= GetFirst as TFarLink;
    while Assigned(FL) do begin
      if aLambda >= (FL.mDist * aNode.mValue)    // если в достижымых пределах
        then Result.Insert(FL.mNodeFar);         // то вставляем в множество
      FL:= GetNext as TFarLink;
    end;
  end;
end;

// Построение множества ограниченных гамм TGamma

function TGraph.GenLimitGams(aLambda: integer): TSet;
var  Node : TNode;
     GammaSet: TSet;
     GammaItem: TGamma;
begin
  Result:= CreateSet;
  Node:= NodeFirst;
```

```
  while Assigned(Node) do begin
    GammaSet:= GenLimitGamma(Node, aLambda);  // формируем огранич. вх. гамму
    GammaItem:= TGamma.Create(Node, GammaSet);// формируем пару вершина+гамма
    Result.Insert(GammaItem);
    Node:= NodeNext;
  end;
end;

// Формирование буфера, содержащего пересечения гамм
// aGams - исходное множество областей достижимости,
// где достижимыми являются отдельные вершины

function TGraph.GenIntersections(aGams: TSet): TBuffer;
var
    GammaItem : TGamma;   // элемент для входящих гамм
        Nodes : TSet;     // достигаемые вершины (TAreal.mSet)
     GammaSet : TSet;     // откуда достигаются (TAreal.mGamma)
         Temp : TSet;     // здесь вычисляется пересечение гамм
      Existed : TSet;     // множество множеств уже обработанных областей
        Areal : TAreal;   // очередной элемент "область"
         Flag : boolean;  // признак пополнения буфера
          Cnt : integer;  // текущее количество элементов в буф. результ.
            i : integer;  // индекс
begin
  Result:= TBuffer.Create;  // буфер результата
  Temp:= CreateSet;           // здесь вычисляется пересечение гамм
  Existed:= CreateSet;        // множество множеств уже обработанных областей
  Nodes:= CreateSet;          // достигаемые вершины (TAreal.mSet)
  GammaSet:= CreateSet;       // откуда достигаются (TAreal.mGamma)
  GammaSet.CopyItems(mNodes); // копия множества всех вершин
  // Создаём и помещаем в буфер элемент-область TAreal такой, что:
  // mNodes - множество достигаемых вершин ПУСТО
  // mGamma - множество вершин, из которых достигается, содержит ВСЕ вершины
  Result.Put(TAreal.Create(Nodes, GammaSet));
  Flag:= false;
  repeat
    Cnt:= Result.GetCount;  // текущее количество элементов в буфере
    for i:=1 to Cnt do begin
      // Цикл по всем элементам буфера
      Flag:= false;                 // сброс признака пополнения буфера
      Areal:= Result.Get as TAreal; // очередной элемент из буфера
      // Перебор всех элементов исходного множества
      GammaItem:= aGams.GetFirst as TGamma;
      while Assigned(GammaItem) do begin
        if not Areal.mSet.Exist(GammaItem.mRoot) then begin
          // Если текущая область ещё не содержит корневой вершины
          // то вычисляем пересечение входящих гамм
          Temp.CopyItems(Areal.mGamma);     // гамма в текущей области
          Temp.Mul(GammaItem.mGamma);       // пересекаем с гаммой вершины
          if Temp.GetCount <> 0 then begin
            // если пересечение не пусто,
            // пытаемя создать и добавить в буфер новую область
            Nodes:= CreateSet;                // достигаемые (TAreal.mSet)
            Nodes.CopyItems(Areal.mSet);      // к существующим вершинам
            Nodes.Insert(GammaItem.mRoot);    // добавляем ещё одну вершину
            if Existed.Exist(Nodes) then begin
              // Если это подмножество достигаемых уже существует в наборе
              Nodes.Free;     // то удаляем дубликат
              //Nodes:= nil;   // и ничего не вставляем
            end else begin
              // Если получено новое подмножество достигаемых вершин
              Existed.Insert(Nodes);     // запоминаем для будущих проверок
```

```
              // создаём новую область и вставляем в буфер
              GammaSet:= CreateSet;    // откуда достигаются (TAreal.mGamma)
              GammaSet.CopyItems(Temp);// копия пересечения GammaSet= Temp
              Result.Put(TAreal.Create(Nodes, GammaSet));
            end; // if
            Flag:= true;  // установить признак пополнения буфера
          end; // if
        end; // if
        GammaItem:= aGams.GetNext as TGamma;  // перебор множества гамм
      end; // while
      // После обработки очередного элемента буфера проверяем:
      // Если обработанный элемент буфера породил хотя бы один новый элемент,
      // то он является подмножеством этого нового и исключается
      // из дальнейшей обработки, а иначе возвращается в буфер
      if Flag then begin
        Existed.Delete(Areal.mSet);// удаляем достигаемые из проверяемых
        Areal.Free;                // удаляем ненужный элемент буфера
      end else begin
        Result.Put(Areal);         // а иначе возвращаем элемент в буфер
      end;
    end; // for
    // выход, если буфер не изменился
  until not Flag and (Cnt=Result.GetCount);
  Temp.Free;
  Existed.Free;
end;


// Поиск множества центров
// с заданной константой проникновения aLambda (дистанцией)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCentLambda(aLambda: integer; aDirect: TCenter): TCostSet;
var   Gams : TSet;
       Buf : TBuffer;
begin
  // Формирование карты дальних указателей (направление карты инверсно):
  case aDirect of
    InCenter:    InitMap(OutCenter);
    OutCenter:   InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование входящих гамм
  // с заданной константой проникновения (дистанцией) aLambda
  Gams:= GenLimitGams(aLambda);
  // Формирование всех непустых пересечений
  Buf:= GenIntersections(Gams);
  // Решение задачи о минимальном покрытии (ЗНП)
  Result:= CollectMinCover(mNodes, Buf);
  // Удаляем результат из буфера
  RemoveItemsFromBuf(Buf, Result.mSet);
  // Освобождаем множество входящих гамм и буфер
  Gams.ClrAndDestroy;
  Gams.Free;
  Buf.ClrAndDestroy;
  Buf.Free;
end;


// Поиск заданного количества P-центров
// с возвращением взвешенного радиуса aLambda
// (обратная задача).
```

```
// aPoly - максимальное количество центров
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPCenters(aPoly: integer;
                            var aLambda: integer;
                            aCenter: TCenter): TCostSet;

var Lambdes : TSet;     // лямбда-список (возрастающее множество расстояний)
       LRec : TSortedNum; // элемент лямбда-списка
     Lambda : integer;    // текущий взвешенный радиус

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества взвешенных расстояний (уникальных)
  procedure LambdasCreate;

    procedure AddLambdas(aNode: TNode);
    var FL : TFarLink;    // очередная дальняя связь
        SN : TSortedNum;  // элемент сортированного списка чисел
        Lambda: integer;  // взвешенное расстояние от текущей вершины
    begin
      with aNode.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          Lambda:= FL.mDist * aNode.mValue;
          SN:= TSortedNum.Create(Lambda);
          if not Lambdes.Insert(SN) then SN.Free;
          FL:= GetNext as TFarLink;
        end;
      end;
    end;

  var  Node : TNode;    // текущая вершина

  begin  { LambdasCreate }
    Lambdes:= CreateSet;
    Node:= NodeFirst;
    while Assigned(Node) do begin   // перебор вершин
      AddLambdas(Node);             // обработать вершину
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPCenters }
  // Формирование карты дальних указателей:
  case aCenter of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Формирование лямбда-списка (возрастающего множества расстояний)
  LambdasCreate;
  // Выбор первого (наименьшего) элемента взвешенного расстояния
  LRec:= Lambdes.GetFirst as TSortedNum;
  repeat
    Lambda:= LRec.mNumber;            // очередное взвешенное расстояние
    Result:= GenPCentLambda(Lambda, aCenter);  // множество центров
    // Если количество центров достигнуто, то выход:
    if Result.mSet.GetCount <= aPoly then Break;
    // а иначе очищаем результат и берём следующий по величине радиус
    Result.ClrAndDestroy;
    Result.Free;
```

```
    Result:= nil;
    // Выбор очередного элемента взвешенного радиуса (по возрастанию)
    LRec:= Lambdes.GetNext as TSortedNum;
  until not Assigned(LRec);
  // Возвращаем взвешенный радиус:
  aLambda:= Lambda;
  // Очистка и уничтожение списка расстояний
  Lambdes.ClrAndDestroy;
  Lambdes.Free;
  // Очистка карты дальних указателей:
  DoneMap;
end;

//#############################################################################
//                                                                            #
//                          P-Медианы                                         #
//                  Кристофидес,  стр. 135                                    #
//                                                                            #
//#############################################################################

// Копирование текущей "медианы" (чёрных вершин)

procedure TGraph.MedianCopy(aRes: TSet);
var N: TNode;
begin
  aRes.Clear;
  N:= NodeFirst;
  while Assigned(N) do begin
    if N.mColor = CBlack then aRes.Insert(N);
    N:= NodeNext;
  end;
end;

// Вычисление стоимости p-подмножества
// (вершины в этот момент уже раскрашены)
// Цвета вершин означают:
// CWhite -- белые ещё не опробованы
// CBlack -- чёрные пробуются в качестве медианных
// CGray  -- серые уже побывали чёрными и не могут быть медианными

function TGraph.MedianCost: integer;
var Node: TNode;       // текущая вершина
    Dist: integer;     // расстояние к ближайшей чёрной
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возвращает расстояние к ближайшей чёрной вершине
  function FindMinDist: integer;
  var FL: TFarLink;     // элемент дальней связи
  begin
    Result:= CInfinity;
    with Node.mFarLinks do begin
      FL:= GetFirst as TFarLink;
      while Assigned(FL) do begin
        if FL.mNodeFar <> Node then begin
          if FL.mNodeFar.mColor = CBlack then begin
            Result:= FL.mDist;
            Break;
          end;
        end;
        FL:= GetNext as TFarLink;
      end;
    end; // with
  end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin  { MedianCost }
  // Перебираем вершины графа (матрицу расстояний),
  // (вклад в стоимость дают только белые и серые вершины)
  Result:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mColor <> CBlack then begin
      // чёрные вершины не дают вклад в стоимость
      Dist:= FindMinDist;  // минимальное расстояние для Node
      Inc(Result, Node.mValue * Dist);   // накопление результата
    end;
    Node:= NodeNext;
  end;
end;

// Поиск медианы "в лоб" перебором
// всех комбинаций медианных подмножеств (демонстрационный метод)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian_A (
                     aPoly: integer;     // кратность (P)
                     aDirect: TCenter;   // внешняя, внутр., внешне-внутр.
                     var aCost: integer  // стоимость (результат)
                    ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивный поиск перебором всех возможных
  // комбинаций чёрных вершин
  // (кандидатов в медианное множество)

  procedure Searching(aIndex: integer { стартовый индекс } );
  var i: integer;
      N: TNode;         // текущий узел
      Cost: integer;    // текущая стоимость p-подмножества
  begin
    // Добавляем последующие вершины
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже оптимальной, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут,
        // погружаемся на следующий уровень
        Inc(Level);       // Level+1
        Searching(i);     // рекурсивный вызов следующего уровня
        Dec(Level);       // Level-1
      end;
      N.mColor:= CGray;   // проверенной вершине назначаем серый цвет
    end;
  end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPMedian_A }
  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать списки ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;                      // начальный уровень в дереве перебора
  NN:= mNodes.GetCount+1-aPoly; // количество узлов, обраб. на каждом уровне
  Searching(1);          // начать перебор с первой вершины
  DoneMap;               // удалить множества дальних указателей
  aCost:= BestCost;      // вернуть цену
end;


// Поиск медианы частичным перебором p-подмножеств (основной метод)
// aDirect = InCenter | OutCenter | InOutCenter

function TGraph.GenPMedian(aPoly: integer;    // кратность
                           aDirect: TCenter;   // внешняя, внутр., внешне-
внутр.
                           var aCost: integer  // стоимость (результат)
                           ): TSet;
var
    BestCost: integer;  // Лучшая цена
    Level : integer;    // Текущий уровень в дереве поиска
    NN: integer;        // количество узлов, обрабатываемых на каждом уровне
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Оценка минимальной стоимости текущего распределения вершин
  // Цвета вершин означают:
  // CWhite -- белые ещё не опробованы
  // CBlack -- чёрные пробуются в качестве медианных
  // CGray  -- серые побывали чёрными и не могут быть медианными

  function EvalBest(aBest: integer): boolean;
  var Node: TNode;       // текущая вершина
        FL: TFarLink;    // элемент дальней связи
       Sum: integer;     // накопитель суммы
      Dist: integer;     // расстояние к ближайшей не серой вершине
         S: TSet;        // неубывающее множество элементов FL
      WhiteCnt: integer; // счётчик белых вершин

    // Поиск ближайшей не серой вершины
    // Node -- текущий не чёрный узел
    function FindMinDist: integer;
    var FL: TFarLink;     // элемент дальней связи
    begin
      Result:= CInfinity;
      with Node.mFarLinks do begin
        FL:= GetFirst as TFarLink;
        while Assigned(FL) do begin
          if (FL.mNodeFar <> Node) and
             (FL.mNodeFar.mColor <> CGray) then begin
            // Можно прикреплять к чёрным и белым
            Result:= FL.mDist;
```

```
          Break;
        end;
        FL:= GetNext as TFarLink;
      end;
    end; // with
  end;

begin  { Eval }
  // Перебираем вершины графа (матрицу расстояний),
  // определяем минимальное расстояние к не серым вершинам,
  // и формируем список из неубывающих чисел (расстояний)
  // (вклад в стоимость дают только белые и серые вершины)

  Sum:= 0;  // накопитель суммы
  // начальное значение счётчика суммируемых белых вершин
  WhiteCnt:= mNodes.GetCount - aPoly;
  S:= CreateSet;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mColor <> CBlack then begin
      // чёрные не дают вклад в стоимость
      Dist:= FindMinDist;  // минимальное расстояние для Node
      case Node.mColor of
        CWhite: begin
                  // взвешенное расстояние к белой вершине
                  // вставляем в неубывающий список
                  FL:= TFarLink.Create(nil, Node, Node.mValue*Dist);
                  S.Insert(FL);
                end;
        CGray:  begin
                  // серую суммируем и уменьшаем счётчик белых
                  Inc(Sum, Node.mValue*Dist); // накопление результата
                  Dec(WhiteCnt);     // счётчик суммируемых вершин -1
                  if Sum>=aBest then Break;
                end;
      end;
    end;
    Node:= NodeNext;
  end;
  // По окончании формирования множества расстояний
  // добавляем WhiteCnt первых из них к накопленному результату
  FL:= S.GetFirst as TFarLink;
  while (WhiteCnt > 0) and (Sum < aBest) do begin
    Inc(Sum, FL.mDist);           // накопление суммы
    FL:= S.GetNext as TFarLink;   // следующий элемент списка
    Dec(WhiteCnt)
  end;
  // Очистка и удаление временного списка:
  S.ClrAndDestroy;
  S.Free;
  Result:= Sum < aBest;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Рекурсивный поиск перебором части комбинаций чёрных вершин
// (кандидатов в медианное множество)

procedure Searching(aIndex: integer { стартовый индекс } );
var i: integer;
    N: TNode;         // текущий узел
    Cost: integer;    // текущая стоимость p-подмножества
begin
  // Добавляем последующие вершины
```

```
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CBlack;
      // Размер p-подмножества достигнут?
      if Level = aPoly-1 then begin
        // Обработка очередного кандидата в медианы
        Cost:= MedianCost;
        if Cost < BestCost then begin
          // Если стоимость ниже оптимальной, то запоминаем результат
          BestCost:= Cost;
          MedianCopy(Result);  // копирование чёрных вершин
        end;
      end else begin
        // Здесь размер p-подмножества не достигнут.
        // Сравним оценку минимальной стоимости с лучшей текущей.
        // Если она меньше текущей, то "ныряем" на уровень вниз
        if  EvalBest(BestCost) then begin
          // погружаемся на следующий уровень
          Inc(Level);    // Level+1
          Searching(i);  // рекурсивный вызов следующего уровня
          Dec(Level);    // Level-1
        end;
      end;
      N.mColor:= CGray;  // проверенной вершине назначаем серый цвет
    end;
    // Перед возвратом на уровень вверх возвращаем белый цвет
    // всем ОБРАБОТАННЫМ вершинам текущего уровня
    for i:= aIndex to NN do begin
      N:= mNodes.GetItem(i + Level) as TNode;  // очередной элемент
      N.mColor:= CWhite;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.GenPMedian }

  // Очистить в вершинах графа поля mColor, mPred:
  ResetNodes;
  // Сформировать множества ближайших вершин:
  case aDirect of
    InCenter   : InitMap(OutCenter);
    OutCenter  : InitMap(InCenter);
    InOutCenter: InitMap(InOutCenter);
  end;
  // Инициализация прочих переменных:
  BestCost:= MaxInt;
  Result:= CreateSet;
  Level:=0;             // текущий уровень в дереве поиска
  NN:= mNodes.GetCount+1-aPoly; // кол-во узлов, обраб. на каждом уровне
  Searching(1);         // начать перебор с первой вершины
  DoneMap;              // удалить множества дальних указателей
  aCost:= BestCost;     // вернуть цену
end;
```

```
//#########################################################################
//                                                                        #
//                ПОКРЫВАЮЩИЕ  ДЕРЕВЬЯ                                    #
//                                                                        #
//#########################################################################

type // Линк для построения покрывающего дерева (остова)

  TTreeLink = class (TLink)
    mMaxTree: boolean;  // false -- миним., true -- макс. остов
    constructor Create(aLink: TLink; aMaxTree: boolean);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;

// Конструируется по данным исходного линка

constructor TTreeLink.Create(aLink: TLink; aMaxTree: boolean);
begin
  inherited Create(aLink.mOwner,  // источник связи
                   aLink.mDest,   // приёмник связи
                   aLink.mValue,  // вес связи
                   true           // нагруженность связей
                   );
  mMaxTree:= aMaxTree;
end;

function TTreeLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  // Для неориентированного графа сравниваем исходную и конечную вершины
  // на предмет встречных линков
  if not mOwner.mOwner.mDirect and  // если не орграф
    (mOwner=(arg as TTreeLink).mDest) and (mDest=(arg as TTreeLink).mOwner)
    then begin
      // Здесь линки направлены встречно, отвергаем дубликат:
      Result:= cmpEq;
      Exit;
    end;
  // Если линки не совпадают, то сортируем по неубыванию веса (длины)
  Result:= cmpLess;
  if mValue > (arg as TTreeLink).mValue then Result:= cmpGreate;
  // Для построения дерева максимального веса результат инвертируем
  if mMaxTree
    then if Result=cmpLess then Result:= cmpGreate else Result:= cmpLess;
end;

// В орграфе выводим по направлению стрелок,
// а в графе - по алфавиту

procedure TTreeLink.Print(var aFile: TextFile);
var N1, N2 : TNode;
begin
  N1:=mOwner;  N2:=mDest;
  if not mOwner.mOwner.mDirect and
     (mOwner.Compare(mDest) = cmpGreate) then begin
    N1:=mDest;   N2:=mOwner;
  end;
  Write(aFile, ' ' + N1.GetName + N2.GetName);
  if mLoadLinks then Write(aFile, '= ', mValue);
end;
```

```
// Добавляе линков, ведущих к ближайшим белым вершинам
// aLinks - накопитель (буфер) линков
// Линки выстраиваются по неубыванию (aMaxTree=false)
// или невозрастанию дистанции (aMaxTree=true)
// Используется для построения остовных деревьев

procedure TNode.AddTreeLinks(aMaxTree: boolean; aLinks: TSet);
var  L : TLink;
    TL : TTreeLink;
begin
  // Обработка исходящих связей
  L:= OutLinkFirst;
  while Assigned(L) do begin
    // Вставляем только линки, ведущие к белым вершинам
    if (L.mDest.mColor = CWhite) then begin
      TL:= TTreeLink.Create(L, aMaxTree);
      if not aLinks.Insert(TL) then TL.Free;
    end;
    L:= OutLinkNext;
  end;
end;

// Алгоритм Прима (Кристофидес, стр. 162)
// Построение минимального (максимального) покрывающего дерева
// Возвращает множество линков TTreeLink  остова

function TGraph.GenCoverTree(aMaxTree: boolean;  // false= min, true = max
                             var aCost: integer  // возвр. стоимость
                            ): TSet;             // множество TTreeLink

var Node: TNode;       // текущая вершина
      TL: TTreeLink;   // текущий линк для дерева
   Links: TSet;        // множество линков для дерева

begin
  aCost:=0;              // накопитель суммы
  Result:= CreateSet;    // множество линков для результата
  Links:= CreateSet;     // множество линков для буфера
  ResetNodes;            // очистка для всех вершин mColor = CWhite
  Node:= NodeFirst;      // исходная вершина
  // Присоединение всех вершин графа (линков на единицу меньше)
  while Assigned(Node) and
       (Result.GetCount < mNodes.GetCount-1) do  begin
    Node.mColor:= CBlack;     // присоединяем текущую к множеству чёрных
    // Добавляем в множество Links линки к ближайшим белым вершинам:
    Node.AddTreeLinks(aMaxTree, Links);
    // Перебираем линки Links в поиске ближайшей белой вершины
    TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
    Node:= nil;                         // искомая вершина пока пуста
    while Assigned(TL) do begin
      Links.Delete(TL);                 // удаляем линк из буфера Links
      // Искомая вершина может быть как источником,
      // так и приёмником связи:
      if TL.mOwner.mColor = CWhite
        then Node:= TL.mOwner   // источник связи
      else if TL.mDest.mColor = CWhite
        then Node:= TL.mDest;   // приёмник связи
      if Assigned(Node) then begin
        // Здесь ближайшая белая вершина найдена:
        Result.Insert(TL);      // вставляем линк в результат
        Inc(aCost, TL.mValue);  // наращиваем стоимость
```

```
        Break;
      end;
      // здесь линк связывает две чёрные вершины:
      TL.Free;                          // удаляем ненужный линк
      TL:= Links.GetNext as TTreeLink; // и берём следующий
    end;
  end;
  // Если остов не построен, то граф не связан
  if Result.GetCount < mNodes.GetCount-1 then begin
    Result.ClrAndDestroy;
    aCost:=0;
  end;
  // Очистка и удаление вспомогательного множества линков:
  Links.ClrAndDestroy;
  Links.Free;
end;


// Поиск минимального ориентированного дерева
// Возвращает корень и вес дерева и само дерево
// Если дерево не существует, возвращаются пустые значения и MaxInt

function TGraph.GenCoverDir(var aRoot: TNode;    // корень
                            var aCost: integer  // вес (стоимость) дерева
                           ): TSet;             // само дерево
var Tree: TSet;     // текущее дерево
    Links: TSet;    // текущее множество линков
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная функция формирует дерево, начиная с вершины aRoot
  // алгоритмом Прима и возвращает его вес (стоимость).
  // Не сформированное дерево очищается

  function RootCover(aRoot: TNode): integer;
  var Node: TNode;       // очередная вершина
      TL : TTreeLink;    // очередной линк
  begin
    Result:= 0;          // вес дерева
    ResetNodes;          // для всех вершин mColor = CWhite
    Node:= aRoot;        // исходная вершина -- корень
    // Пытаемся присоединенить все вершины графа (линков на единицу меньше)
    while Assigned(Node) and
          (Tree.GetCount < mNodes.GetCount-1) do  begin
      Node.mColor:= CBlack;     // текущую в множество чёрных
      // добавляем линки к ближайшим белым вершинам:
      Node.AddTreeLinks(false, Links);
      // Перебираем линки Links в поиске ближайшей белой вершины
      TL:= Links.GetFirst as TTreeLink;   // первый линк = кратчайший
      Node:= nil;                         // искомая вершина пока пуста
      while Assigned(TL) do begin         // пока существуют линки
        Links.Delete(TL);                 // удаляем линк из буфера Links
        // Искомая вершина может быть только приёмником связи:
        if TL.mDest.mColor = CWhite then begin
          // Здесь ближайшая белая вершина найдена:
          Node:= TL.mDest;        // будет обработана в следующем цикле
          Tree.Insert(TL);        // вставляем линк в результат
          Inc(Result, TL.mValue); // наращиваем стоимость
          Break;
        end;
        // здесь линк связывает две чёрные вершины:
        TL.Free;                          // удаляем ненужный линк
        TL:= Links.GetNext as TTreeLink; // и берём следующий
      end;
```

```
    end;
    Links.ClrAndDestroy; // очистка оставшихся в очереди линков
    // Если остов не построен, очищаем накопитель линков
    if Tree.GetCount < mNodes.GetCount-1 then begin
      Tree.ClrAndDestroy;  // очистка поддерева
      Result:= MaxInt;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var Root: TNode;    // очередной корень дерева
    Cost: integer;  // стоимость (вес) дерева

begin { TGraph.GenCoverDir }
  Result:=nil;  aRoot:= nil;
  aCost:= MaxInt;
  Links:= CreateSet;  // текущее множество линков
  Tree:= CreateSet;   // текущее остовное дерево
  // Перебор всех вершин:
  Root:= NodeFirst;
  while Assigned(Root) do begin
    Cost:= RootCover(Root);
    // Если дерево построено:
    if Tree.GetCount > 0 then begin
      // Если стоимость меньше максимальной
      if Cost<aCost then begin
        // то сохраняем результат
        if Assigned(Result)
          then Result.ClrAndDestroy // удаление существующих линков
          else Result:=CreateSet;   // создание копии дерева
        Result.CopyItems(Tree);     // копируем дерево (линки)
        Tree.Clear;                 // удаляем сохранённые линки
        aRoot:= Root;               // запоминаем корень
        aCost:= Cost;               // запоминаем стоимость (вес)
      end else begin
        Tree.ClrAndDestroy;  // очистка дерева, если оно не минимально
      end;
    end;
    Root:= NodeNext;
  end;
  // Удаление рабочих буферов:
  Tree.Free;
  Links.Free;
end;

// Построение всех остовных деревьев (Кристофидес, стр. 149)
// для неориентированного графа

procedure TGraph.ExpoAllCovers;
var
  Links: TSet;       // множество всех линков графа
   Tree: TSet;       // множество линков дерева
  Level: integer;    // текущий уровень
     NN: integer;    // количество линков, обрабатываемых на уровне
    Cnt: integer;    // счётчик остовных деревьев
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  procedure Local_Init;
  var Node: TNode;   // текущая вершина
  begin
    ResetNodes;          // для всех вершин mColor=0, mPred=nil
    Tree:= CreateSet;    // множество для остовного дерева
    Links:= CreateSet;   // множество для всех линков
    Node:= NodeFirst;
```

```
    while Assigned(Node) do begin
      with Node do begin
        mRoot:= Node;          // корень поддерева
        AddTreeLinks(false, Links);  // добавление линков вершины
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Вспомогательная функция вычисления стоимости остова
  // как суммы весов его рёбер

  function Cost(aTree: TSet): integer;
  var TL: TTreeLink;  // текущий линк для дерева
  begin
    Result:= 0;
    TL:= aTree.GetFirst as TTreeLink;
    while Assigned(TL) do begin
      Inc(Result, TL.mValue);
      TL:= aTree.GetNext as TTreeLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Объединение двух поддеревьев
  // aTL.mLink.mOwner -- принадлежит поддереву T1
  // aTL.mLink.mDest  -- принадлежит поддереву T2
  function Union(aTL: TTreeLink): boolean;
  var   N2: TNode;    // узел в T2
        p2: TNode;    // предшествующий узел
        tmp: TNode;   // временное хранилище
  begin
    with aTL do begin
      // Если концы линка принадлежат одному поддереву, то выход
      Result:= mOwner.mRoot <> mDest.mRoot;
      if not Result then Exit;
      // Подготовка к реверсу обратных ссылок mPred
      N2:= mDest;         // текущая вершина в поддереве T2
      p2:= N2.mPred;      // предшествующая вершина в T2
      N2.mPred:=mOwner;   // новая предшествующая взята из T1
      // Реверсирование обратных ссылок:
      while Assigned(p2) do begin
        tmp:= p2.mPred;   // временно сохранить ссылку в T2
        p2.mPred:= N2;    // обновить обратную ссылку
        N2:= p2;          // сдвинуть вершину по направлению к 2-му корню
        p2:= tmp;         // восстановить ссылку в T2
      end;
      // Замена 2-го корня на 1-й:
      // root1 = mOwner.mRoot.Expo
      // root2 = mDest.mRoot.Expo
      p2:= mDest.mRoot as TNode;    {###}
      N2:= NodeFirst;
      while Assigned(N2) do begin
        if N2.mRoot = p2 then N2.mRoot:= mOwner.mRoot;
        N2:= NodeNext;
      end;
    end;  // with
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Удаление ребра и расщепление дерева
  procedure Remove(aTL: TTreeLink);
  var Root: TNode;  // Новый корень для 2-го поддерева
      S1 : TSet;    // внутренний слой вершин
```

```
    S2 : TSet;    // внешний слой вершин
    Node: TNode;
begin
  S1:= CreateSet;    S2:= CreateSet;
  // Выбираем новый корень для 2-го поддерева
  with aTL do if mOwner.mPred = mDest
    then Root:= mOwner    // источник связи
    else Root:= mDest;    // приёмник связи
  Root.mRoot:= Root;      // пометка корня самим собой
  Root.mPred:= nil;       // обратная ссылка в корне пуста
  S1.Insert(Root);        // организуем нулевой слой из корневой вершины
  repeat
    // Накопление внешнего слоя:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // если примыкает к внутреннему слою, то добавляем к внешнему
      if S1.Exist(Node.mPred) then begin
        Node.mRoot:= Root;
        S2.Insert(Node);
      end;
      Node:= NodeNext;
    end;
    // Выход, если внешний слой пуст
    if S2.GetCount=0 then Break;
    // Копирование внешнего слоя во внутренний и его очистка
    S1.CopyItems(S2);
    S2.Clear;
  until false;
  S1.Free; S2.Free;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Рекурсивный перебор всех подмножеств
// (комбинаций) из L по N-1 линков
// L - количество рёбер или дуг графа = Links.GetCount
// N - количество вершин графа = mNodes.GetCount

procedure Searching(aIndex: integer { стартовый индекс } );
var  i: integer;
    TL: TTreeLink;  // текущий линк для дерева
begin
  // Добавляем последующие вершины
  for i:= aIndex to NN do begin
    TL:= Links.GetItem(i + Level) as TTreeLink;  // очередной линк
    if Union(TL) then begin
      // Этот линк не создаёт цикла, вставляем в остов
      Tree.Insert(TL);
      if Level = mNodes.GetCount-2 then begin
        // Остов сформирован
        Inc(Cnt);      // счётчик остовных деревьев
        Write(Cnt:4, '':2, Cost(Tree):3, '':2);
        Tree.Expo;
      end else begin
        // Здесь размер подмножества не достигнут, остов не сформирован,
        // погружаемся на следующий уровень
        Inc(Level);    // Level+1
        Searching(i);  // рекурсивный вызов следующего уровня
        Dec(Level);    // Level-1
      end;
      // Возвращаем предыдущее состояние поддеревьев
      Tree.Delete(TL);  // удаляем из остовного множества
      Remove(TL);       // и расщепляем дерево
    end; // if
```

```
    end; // for
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
begin { TGraph.ExpoAllCovers }
  Local_Init;
  Cnt:= 0;      // счётчик остовных деревьев
  Level:=0;     // текущий уровень
  // количество линков, обрабатываемых на каждом уровне:
  // (mNodes.GetCount - 1) = количество рёбер дерева
  NN:= Links.GetCount + 1 - (mNodes.GetCount - 1);
  Searching(1);
  Links.ClrAndDestroy;
  Links.Free;
end;

//#########################################################################
//                                                                        #
//                    ПОТОКОВЫЕ  АЛГОРИТМЫ                                #
//                                                                        #
//#########################################################################

// Вычисление максимального потока
// при условии отсутствия минимальных потоков (для всех дуг mLow=0)
// Майника Э. Алгоритмы оптимизации на сетях и графах, 4.2 (стр. 91)

function TGraph.CalcMaxFlow0(aSource, aDest: TNode): integer;
var Delta: integer;   // очередное приращение потока
    Que: TBuffer;   // очередь вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт начального потока
  // на случай, когда заданы минимальные потоки в сети (mLow>0)

  function CalcStartFlow: integer;
  var Link: TLink;    // исходящая дуга
  begin
    Result:= 0;
    Link:= aSource.OutLinkFirst;
    while Assigned(Link) do begin
      Inc(Result, Link.mFlow);
      Link:= aSource.OutLinkNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Построение увеличивающей цепи
  // с подсчётом соответствующего приращения потока

  function CalcDelta: integer;
  var Node: TNode;
      Link: TLink;    // исходящая или входящая дуга
  begin
    Result:=0;
    // установка mColor=0, mPred=nil, mDist= MaxInt, mFlow= MaxInt
    ResetNodes;
    Que.Clear;   // очистка очереди вершин
    // Пометку вершин начинаем с источника
    aSource.mColor:= CGray;
    Que.Put(aSource);
    while Que.GetCount>0 do begin
      Node:= Que.Get as TNode;
      // Выход из цикла, если помечен сток
      if Node=aDest then Break;
      // Обработка исходящих связей текущего узла:
```

```
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Помечаем только белые вершины, достигаемые через ненасыщенные дуги
        if (Link.mDest.mColor=CWhite)            // если вершина не помечена
           and (Link.mFlow < Link.mHigh) then begin  // и дуга не насыщена
          Link.mDirect:= +1;  // это увеличивающаяся дуга
          // помечаем концевую вершину и ставим её в очередь:
          with Link.mDest do begin
            mColor:= CGray;     // цвет концевой вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mHigh - Link.mFlow); // поток
          end;
          Que.Put(Link.mDest); // в очередь приёмник дуги
        end;
        Link:= Node.OutLinkNext;
      end;
      // Обработка входящих связей текущего узла:
      Link:= Node.InLinkFirst;
      while Assigned(Link) do begin
        // Помечаем белые вершины, из которых поступает ненулевой поток
        if (Link.mOwner.mColor=CWhite)          // если вершина не помечена
           and (Link.mFlow > Link.mLow) then begin // и поток можно уменьшить
          Link.mDirect:= -1;  // это уменьшающая дуга
          // помечаем вершину (источник дуги) и ставим её в очередь:
          with Link.mOwner do begin
            mColor:= CGray;     // цвет исходной вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner); // в очередь источник дуги
        end;
        Link:= Node.InLinkNext;
      end;  // while
    end;  // while
    // Если сток не помечен, то увеличивающая цепочка не найдена
    // и приращение потока равно нулю
    if aDest.mColor = CWhite then Exit;
    // Здесь  увеличивающая цепочка найдена
    Result:= aDest.mFlow; // допустимое увеличение потока
    // Обратное движение от стока к истоку с корректировкой потока
    Node:= aDest;  // сток
    while Node <> aSource do begin
      with Node.mLink do Inc(mFlow, mDirect * Result); // mDirect = +1 / -1
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin  { TGraph.CalcMaxFlow0 }
  Result:= 0;
  if not Assigned(aSource) or not Assigned(aDest) then Exit;
  Result:= CalcStartFlow; // начальный поток
  Que:= TBuffer.Create;   // рабочая очередь вершин
  // Повторяем поиск увеличивающих цепей, пока это возможно
  repeat
    Delta:= CalcDelta;    // подсчёт приращения потока
    Inc(Result, Delta);   // накопление результата
  until Delta=0;
  Que.Free;
end;
```

```
// Вычисление минимального потока
// (когда минимальное значение потока больше нуля)
// Басакер Р. Саати Т. Конечные графы и сети, стр.327 (7.8)

function TGraph.CalcMinFlow(aSource, aDest: TNode): integer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Восстановление данных о потоке после удаления искусственных вершин и дуг
  // arg = false -- минимальный поток не существует
  // arg = true  -- минимальный поток найден

  procedure RestoreFlowData(arg: boolean);
  var Node: TNode;
      Link: TLink;      // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        // Перебор исходящих линков вершины:
        Link:= OutLinkFirst;
        while Assigned(Link) do begin
          with Link do begin
            mLow:= mTemp;           // минимум
            mHigh:= mHigh + mLow;   // максимум
            // текущий минимальный поток:
            if arg
              then mFlow:= mFlow + mLow  // в случае успеха
              else mFlow:= 0;            // если поток не существует
          end; // with
          Link:= OutLinkNext;
        end; // while
      end; // with
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;       // текущая вершина
    Link: TLink;       // текущий линк
    Reverse: TLink;    // обратный линк T->S во вспомогательном графе
    Flow: integer;     // максимальный поток
    SumLow: integer;   // сумма всех минимумов
    Sx: TNode;         // вспомогательный исток
    Tx: TNode;         // вспомогательный сток
    OldLink: TLink;    // линк T->S в исходном графе
    OldLow, OldHigh: integer; // сохранённые данные обратного линка

begin    { TGraph.CalcMinFlow }

  // Если не указаны вершины, то выход
  if not Assigned(aSource) or not Assigned(aDest) then begin
    Result:=-1;
    Exit;
  end;

  // Создаём:
  Sx:= TNode.Create(0, Self); // вспомогательный исток
  Tx:= TNode.Create(0, Self); // вспомогательный сток
  SumLow:=0;    // здесь накапливаем сумму всех минимальных потоков
  // Первый перебор вершин с подсчётом сумм исходящих минимальных потоков:
  Node:= NodeFirst;
```

```
  while Assigned(Node) do begin
    Node.mFlow:=0; // сумма исходящих минимумов
    with Node do begin
      // Перебор исходящих линков вершины:
      Link:= OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          Inc(Node.mFlow, mLow);  // сумма исходящих минимумов
          mHigh:= mHigh - mLow;   // пересчёт максимума
          mTemp:= mLow;           // временно сохраняем минимум
          mLow:= 0;               // и обнуляем его
          mFlow:=0;               // обнуляем поток
        end;
        Link:= OutLinkNext;
      end;
    end; // with
    Node:= NodeNext;
  end; // while

  // Второй перебор вершин с подсчётом сумм входящих минимальных потоков:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      // Перебор входящих линков вершины:
      Link:= InLinkFirst;
      while Assigned(Link) do begin
        Dec(Node.mFlow, Link.mTemp);   // минус сумма исходящих минимумов
        Link:= InLinkNext;
      end;
    end;
    // Если вершина является источником или стоком:
    if Node.mFlow <> 0 then begin
      // Если вершиная является источником:
      if Node.mFlow > 0 then begin
        // создаём дугу из вершины во вспомогательный сток
        Link:= Node.MakeLink(Tx, 0);  // создаём линк
        Link.mHigh:= Node.mFlow;      // с данными о потоке
      end else begin
        // создаём дугу из вспомогательного истока в вершину
        Link:= Sx.MakeLink(Node, 0);  // создаём линк
        Link.mHigh:= -Node.mFlow;      // с данными о потоке
      end
    end;
    Inc(SumLow, Abs(Node.mFlow)); // двойная сумма всех минимальных потоков
    Node:= NodeNext;
  end; // while
  SumLow:= SumLow div 2;  // одинарная сумма всех минимальных потоков

  // Если сумма минимальных потоков = 0, то выходим из процедуры
  if SumLow=0 then begin
    // Освобождаем:
    Sx.Free;     // вспомогательный исток
    Tx.Free;     // вспомогательный сток
    // Восстанавливаем данные о потоках:
    RestoreFlowData(false);
    Result:=0;
    Exit;
  end;

  // Вставляем вспомогательные вершины в граф:
  mNodes.Insert(Sx);   // исток
  mNodes.Insert(Tx);   // сток
```

```
// Соединяем обратной дугой сток с истоком
// (поток в обратной дуге не ограничен):

// На случай отсутствия обратной дуги:
OldLow:= 0; OldHigh:=0;

// Обратная дуга уже существует?
OldLink:= aDest.GetLink(aSource);
if Assigned(OldLink) then begin
  // да, назначаем её в качестве реверсивной и запоминаем параметры
  Reverse:= OldLink;    // реверсивная дуга
  with OldLink do begin
    // да, запоминаем данные потока
    OldLow := mLow;     // нижний предел
    OldHigh:= mHigh;    // верхний предел
    // и устанавливаем новые
    mLow := 0;          // нижний предел отсутствует
    mHigh:= MaxInt;     // неограниченная пропускная способность
  end // with
end else begin
  // здесь обратной дуги нет, поэтому создаём новую дугу
  Reverse:= aDest.MakeLink(aSource, 0); // дуга
  Reverse.mHigh:= MaxInt;   // с максимальной пропускной способностью
end;

// Во вновь построенном вспомогательном графе
// вычисляем максимальный поток между вспомогательными истоком и стоком

Flow:= CalcMaxFlow0(Sx, Tx);

// После этого минимальный поток протекает через реверсную дугу:
Result:= Reverse.mFlow;  // запоминаем
Reverse.mFlow:= 0;       // и обнуляем

// Удаляем обратную дугу, или восстанавливаем предыдущую, если она была

// Обратная дуга существовала?
if Assigned(OldLink) then with OldLink do begin
  // да восстанавливаем
  mLow := OldLow;
  mHigh:= OldHigh;
end else begin
  // Дуга в исходном графе не существовала,
  // разрываем обратную связь между стоком и истоком
  aDest.RemoveLink(aSource);
end;

// Удаляем из графа и освобождаем:

RemoveNode(Sx);  Sx.Free;     // вспомогательный исток
RemoveNode(Tx);  Tx.Free;     // вспомогательный сток

// Если вспомогательный максимальный поток (Sum)
// равен сумме минимальных потоков (SumLow)
// то минимальный поток в исходном графе существует

if Flow = SumLow then begin
  // вспомогательные дуги насыщены
  RestoreFlowData(true); // формируем новые данные о потоках
end else begin
  // вспомогательные дуги НЕ насыщены
```

```
    Result:=-1;              // минимального потока не существует
    RestoreFlowData(false); // восстанавливаем прежние данные о потоках
  end;

end;


// Вычисление максимального потока
// когда минимальный поток может быть ненулевым

function TGraph.CalcMaxFlow(aSource, aDest: TNode): integer;
begin
  // Сначала вычисляем минимальный поток:
  Result:= CalcMinFlow(aSource, aDest);
  // Если он не существует, то не существует и максимального:
  if Result < 0 then Exit;
  // После формирования минимального потока формируем максимальный:
  Result:= CalcMaxFlow0(aSource, aDest);
end;


// Вычисление минимальной стоимости заданного потока
// Майника, стр. 104
// Кристофидес, стр. 339

function TGraph.CalcMinCostFlow(aSource, aDest: TNode; // источник и сток
                                aFlow: integer         // величина потока
                                ): integer;            // стоимость потока

var  Gray: TSet;      // множество серых (купленных) вершин
     Que: TBuffer;  // очередь вершин используется в CalcDeltaFlow
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка данных потока и вершинных чисел,
  // вызывается единожды в начале метода

  procedure ClearFlowData;
  var Node: TNode;
      Link: TLink;      // исходящая дуга
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node.mDist:= 0;      // очищаем вершинные числа
      // Перебор исходящих линков вершины:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mFlow:= 0;          // текущий поток = 0
        Link.mColor:= CWhite;    // цвет дуги белый
        Link:= Node.OutLinkNext;
      end; // while
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета (mColor) и окраска вершины-истока
  // Вызывается в начале поиска очередной увеличивающей цепочки

  procedure ClearColors;
  var Node: TNode;      // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
```

```
      Node.mColor:= CWhite;
      Node.mFlow:= MaxInt; // поток через вершины не ограничен
      Node:= NodeNext;
    end;
    // Красим вершину-исток и помещаем в множество серых вершин:
    aSource.mColor:= CGray;
    Gray.Clear;
    Gray.Insert(aSource);
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Попытка приобретения очережной прямой дуги
  // Исходные данные:
  // - текущее множество серых вершин Gray
  // - текущий поток в дугах
  // - текущее состояние вершинных чисел
  // Возвращает true в случае покупки дуги изменением вершинных чисел

  function Buying: boolean;
  var Node: TNode;       // текущая вершина
      Link: TLink;       // текущий линк
      Delta: integer;    // очередное приращение суммы
      MinDelta: integer;// минимальное приращение суммы
  begin
    MinDelta:= MaxInt;
    // Перебор серых вершин:
    Node:= Gray.GetFirst as TNode;
    while Assigned(Node) do begin
      // Просмотр исходящих дуг:
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mDest.mColor = CWhite then begin
          // Если поток не насыщен:
          if mFlow < mHigh then begin
            // Прицениваемся к увеличивающей дуге:
            // Link.mValue - цена увеличивающей дуги
            // mOwner.mDist - сумма оплаты цепочки вплоть до Node
            // mDest.mDist - частичная плата вплоть до mDest
            // Delta - то, что нужно доплатить за приобретение дуги Link
            Delta:= mValue - (mDest.mDist - mOwner.mDist);
            if Delta < MinDelta
              then MinDelta:= Delta;  // запоминаем минимальную доплату
          end; // if
        end;
        Link:= Node.OutLinkNext;
      end; // while
      Node:= Gray.GetNext as TNode; // Следующая вершина серого множества
    end; // while Assigned(Node)
    Result:= MinDelta < MaxInt;     // true, если дуга куплена
    if not Result or (MinDelta=0) then Exit;
    // Наращиваем mDist - вершинные числа белых вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mColor = CWhite then Inc(Node.mDist, MinDelta);
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск потока, проходящего по уже купленным дугам
  // Исходные данные:
  // - текущее множество серых вершин
  // - текущий поток
  // - текущие вершинные числа
```

```
// Расширяет множество серых вершин.
// Возвращает допустимое приращение потока (или ноль)
// и стоимость единицы потока

function CalcDeltaFlow(var aCost: integer): integer;
var
    Node: TNode;      // текущая вершина
    Link: TLink;      // исходящие и входящие дуги
begin
  Result:=0; aCost:=0;
  Que.Clear;  // очищаем очередь вершин
  // Ставим в очередь серые вершины
  Node:= Gray.GetFirst as TNode;
  while Assigned(Node) do begin
    Que.Put(Node);
    Node:= Gray.GetNext as TNode;
  end;
  Que.Put(aSource);
  // Обработка вершин из очереди:
  while Que.GetCount>0 do begin
    Node:= Que.Get as TNode;
    // Обработка исходящих дуг текущего узла:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do if mDest.mColor=CWhite then begin
        // если вершина не помечена
        // Помечаем только белые вершины,
        // достигаемые через купленые ненасыщенные дуги
        // Link.mValue -- стоимость дуги
        // Link.mDest.mDist - mOwner.mDist -- разность вершинных чисел
        if (mValue = mDest.mDist - mOwner.mDist) // дуга куплена
           and (mFlow < mHigh) then begin  // и дуга не насыщена
          mDirect:= +1;  // отмечаем прямую дугу
          // помечаем концевую вершину и ставим в очередь:
          mDest.mColor:= CGray;     // цвет концевой вершины серый
          mDest.mPred:= Node;       // предшествующая вершина
          mDest.mLink:= Link;       // линк на предшествующую вершину
          mDest.mFlow:= Minimum(Node.mFlow, mHigh - mFlow); // поток
          Que.Put(mDest);       // в очередь приёмник дуги
          Gray.Insert(mDest);  // пополняем множество окрашенных
        end;
      end; // if
      Link:= Node.OutLinkNext;
    end;
    // Обработка входящих связей текущего узла:
    Link:= Node.InLinkFirst;
    while Assigned(Link) do begin
      with Link do if mOwner.mColor=CWhite then begin
        // Помечаем только белые вершины,
        // из которых поступает ненулевой поток по купленым дугам
        if mFlow > 0 then begin  // поток в дуге можно уменьшить
          mDirect:= -1;  // отмечаем встречную дугу
          // помечаем вершину и ставим в очередь:
          with Link.mOwner do begin
            mColor:= CGray;     // цвет исходной вершины серый
            mPred:= Node;       // предшествующая вершина
            mLink:= Link;       // линк на предшествующую вершину
            mFlow:= Minimum(Node.mFlow, Link.mFlow); // допустимый поток
          end;
          Que.Put(Link.mOwner);      // в очередь источник дуги
          Gray.Insert(Link.mOwner); // пополняем множество окрашенных
        end;
```

```
        end;
        Link:= Node.InLinkNext;
      end;  // while Assigned(Link)
      // Если помечен сток, то выход из цикла:
      if aDest.mColor <>  CWhite then Break;
    end;  // while Que.GetCount>0
    // Если сток помечен, то увеличивающая цепочка найдена
    // возвращаем допустимое увеличение потока и цену единицы потока
    if aDest.mColor <> CWhite then begin
      Result:= aDest.mFlow;
      aCost:= aDest.mDist;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Пересчёт потока вдоль предварительно найденной цепочки
  // aDelta - приращение потока

  procedure RecalcFlow(aDelta: integer);
  var Node: TNode;     // текущая вершина
  begin
    // Обратное движение от стока к истоку с корректировкой потока в дугах
    Node:= aDest;  // сток
    // Цикл, пока не достигнем истока:
    while Node <> aSource do begin
      // Node.mLink - линк из предшествующей вершины
      // Node.mLink.mDirect = +1 / -1
      with Node.mLink do Inc(mFlow, mDirect * aDelta); // поток
      Node:= Node.mPred;  // предшествующая вершина
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
  Flow: integer;         // накопленный поток
  DeltaFlow: integer;    // приращение потока
  Cost1: integer;        // цена единицы потока
  OK: boolean;           // признак приобретения дуги

begin { TGraph.CalcMinCostFlow }

  Result:= 0;        // накопленная стоимость потока
  Flow:= 0;          // накопленный поток
  Gray:= CreateSet; // множество окрашенных (купленных) вершин
  Que:= TBuffer.Create;  // создаём очередь вершин
  ClearFlowData;     // очистка потока в дугах (mFlow=MaxInt)
  ClearColors;       // очистка цвета и начальная установка серого множества
  OK:= false;        // признак приобретения дуги

  // Пока поток не достиг заданного значения:
  while Flow < aFlow do begin
    // Пытаемся "купить" увеличивающую цепочку (дугу)
    // и формируем вершинные числа.
    // Если куплена хотя бы одна дуга, пытаемся провести поток:
    repeat
      // Пытаемся провести поток и определить цену единицы потока
      DeltaFlow:= CalcDeltaFlow(Cost1);
      if DeltaFlow > 0  then begin
        OK:= true;
        // Поток возможен, ограничиваем его приращение:
        if DeltaFlow > aFlow - Flow then DeltaFlow:= aFlow - Flow;
        // Распределяем поток по дугам и и определяем цену единицы потока:
        RecalcFlow(DeltaFlow);         // пересчёт потока в дугах
        Inc(Flow, DeltaFlow);          // суммарный поток
```

```
        Inc(Result, Cost1*DeltaFlow); // суммарная стоимость потока
        ClearColors;                  // очистка цвета и серого множества
      end else begin
        // Если поток не проведен, пытаемся купить хотя бы одну дугу
        // (функция Buying модифицирует вершинные числа)
        OK:= Buying;
        // Если цепочку купить нельзя, то заданный поток не существует
        if not OK then Break;
      end;
      // пока поток не достигнет нужного, либо не найдено приращение
    until Flow = aFlow;
    // Если цепочку купить нельзя, то заданный поток не существует
    if not OK then begin
      Result:=-1;  // стоимость = -1
      Break;
    end;
  end; // while Flow < aFlow
  // Очистка памяти:
  Gray.Free; // серое множество
  Que.Free;  // рабочая очередь
end;


//############################################################################
//                                                                           #
//                  П А Р О С О Ч Е Т А Н И Я                                #
//                                                                           #
//############################################################################
{ Варианты формирования паросочетаний:

  TPairs = (pMaxN,  // максимальной мощности
            pMaxW   // максимального веса (стоимости)
            pMinW   // минимального веса (стоимости)
           );
}

type  // TPair -- Ребро представлено парой вершин

     TPair = class (TCostSet)
       function Compare(arg: TItem): TCompare;  override;
       procedure Print(var aFile: TextFile); override;
     end;

function TPair.Compare(arg: TItem): TCompare;
begin
  if Self = arg then begin Result:= cmpEq; Exit; end;
  if mCost = (arg as TCostSet).mCost then begin
    // При равной цене сравнением множества
    Result:= mSet.Compare((arg as TCostSet).mSet);
  end else begin
    // Если цены не совпали, сравниваем по цене (дешёвые - в начало)
    if mCost > (arg as TCostSet).mCost
      then Result:= cmpGreate
      else Result:= cmpLess
  end;
end;

// Вывод множества вершин (у пары их будет две)

procedure TPair.Print(var aFile: TextFile);
var N: TNode;
    k: integer;
```

```
begin
  k:= mSet.GetCount;
  N:= mSet.GetFirst as TNode;
  while Assigned(N) do begin
    Write(aFile, N.GetName);
    Dec(k);
    if k > 0 then Write(aFile, '-');
    N:= mSet.GetNext as TNode;
  end;
  Write(aFile, ' = ', mCost);
end;

type  // TPairsBlock -- Блок, содержащий пары

      TPairsBlock = class (TSetList)
        mLabel : TNode;    // метка блока (центральная вершина)
        mCost  : integer;  // вес наиболее лёгкой пары в блоке
        function Compare(arg: TItem): TCompare; override;
        procedure Print(var aFile: Text); override;
        destructor Destroy;  override;
      end;

destructor TPairsBlock.Destroy;
begin
  ClrAndDestroy;  // очистка с уничтожением элементов
  Inherited;
end;

// Блоки сортируются в порядке неубывания размера блока

function TPairsBlock.Compare(arg: TItem): TCompare;
begin
  Result:= cmpLess;
  if GetCount > (arg as TPairsBlock).GetCount  then Result:= cmpGreate
end;

procedure TPairsBlock.Print(var aFile: Text);
begin
  Writeln(aFile,mLabel.GetName,' Cost=', mCost:3,'  Count=',GetCount:2);
  inherited Print(aFile);
  Writeln(aFile,'- - - - -');
end;


// Метод генерации паросочетаний (п/с) трёх видов:
// pMaxN -- максимальной мощности (без учёта веса рёбер)
// pMaxW -- максимального веса
// pMinW -- минимального веса

function TGraph.GenPairs(aMode: TPairs): TCostSet;

var Blocks: TSet;         // Блоки пар (рёбер)
    BestCost: integer;    // Текущая лучшая (наименьшая) сумма
    BestCnt: integer;     // Текущая лучшая (наибольшая) мощность п/с
    Pairs: TCostSet;      // Текущее паросочетание
    Nodes: TSet;          // Текущее множество присоединённых вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инициализация вершин и дуг
  procedure InitNodes;
  var Node: TNode;
      LD, LR : TLink; // прямой и обратный линки
  begin
```

```
  // Предварительная очистка поля мощности
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mPower:= 0;  // здесь будут подсчитываться смежные линки
    Node:= NodeNext;
  end;
  // Перебор всех линков с целью отбора наименьших или наибольших
  // (в случае наличия встречных линков)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do begin
      // Красим вершину белым
      mColor:= CWhite;
      // Перебор прямых линков LD
      LD:= OutLinkFirst;
      while Assigned(LD) do begin
        // Находим LR -- встречный линк
        LR:= LD.mDest.GetLink(Node);
        if Assigned(LR) then begin
          // Встречный линк найден
          if (aMode=pMaxW) and (LR.mValue < LD.mValue) then begin
            // При поиске максимума берём больший
            LD.mColor:= CBlack; // будет обрабатываться
            LR.mColor:= CWhite; // НЕ будет обрабатываться
          end else begin
            // При поиске минимума берём меньший
            LR.mColor:= CBlack; // будет обрабатываться
            LD.mColor:= CWhite; // НЕ будет обрабатываться
          end;
        end else begin
          // встречный линк НЕ найден
          LD.mColor:= CBlack; // будет обрабатываться прямой линк
        end;
        // Подсчёт инцидентных линков в двух соседних вершинах
        Inc(LD.mOwner.mPower);
        Inc(LD.mDest.mPower);
        LD:= OutLinkNext;
      end;
    end;
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск вершины, способной дать блок максимального размера

function GetBestNode: TNode;
var Node: TNode;
    Max : integer;
begin
  Max:= 0; Result:= nil;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    with Node do if (mColor=CWhite)    // вершина ещё не обработана
                 and (mPower > Max) // и её текущая мощность больше
    then begin
      Result:= Node;  // очередная лучшая вершина
      Max:= mPower;   // текущее число смежных чёрных линков
    end;
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
// Формирование всех возможных пар вершин (рёбер)
// и группировка их в блоки

function GenBlocks: TSet;

  // Создание пары и вставка её в блок:
  function MakePair(aLink: TLink): TPair;
  begin
    Result:= TPair.CreateEmpty;
    with aLink do begin
      // В множество пары вставляем смежные с линком вершины:
      Result.mSet.Insert(mOwner);
      Result.mSet.Insert(mDest);
      // Уменьшаем мощности соседних вершин:
      Dec(mOwner.mPower);
      Dec(mDest.mPower);
    end;
    // Цену (вес) пары формируем в зависимости от варианта п/с
    case aMode of
      pMaxN: Result.mCost:=  1;              // наибольшее
      pMinW: Result.mCost:=  aLink.mValue;   // минимального веса
      pMaxW: Result.mCost:= -aLink.mValue;   // максимального веса ***
    end;
  end;

var Node: TNode;          // текущая вершина
    Link: TLink;          // текущий исходящий линк
    Pair: TPair;          // текущая пара
    Min: integer;         // наименьший вес пары в блоке
    Block: TPairsBlock;   // текущий блок

begin  { GenBlocks }
  InitNodes;            // инициализация полей mPower
  Result:= CreateSet;   // множество блоков
  // Перебор вершин:
  Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
  while Assigned(Node) do begin
    Min:= MaxInt;     // здесь формируется вес самой лёгкой пары в блоке
    Block:=nil;       // текущий блок пока не создан

    // Перебор исходящих линков:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      // Если линк не обработан
      if Link.mColor = CBlack then begin
        Link.mColor:= CWhite;  // восстанавливаем цвет
        Pair:= MakePair(Link); // создаём пару
        // Запоминаем вес легчайшей пары в блоке
        if Min > Pair.mCost then Min:= Pair.mCost;
        // Если блок ещё не создан, то создаём:
        if not Assigned(Block) then Block:= TPairsBlock.Create;
        Block.Insert(Pair);  // вставляем пару в блок
      end;
      Link:= Node.OutLinkNext;  // следующий линк
    end; // while

    // Перебор входящих линков:
    Link:= Node.InLinkFirst;
    while Assigned(Link) do begin
      // Если линк не обработан
      if Link.mColor = CBlack then begin
        Link.mColor:= CWhite;  // восстанавливаем цвет
```

```
        Pair:= MakePair(Link); // создаём пару
        // Запоминаем вес легчайшей пары в блоке
        if Min > Pair.mCost then Min:= Pair.mCost;
        // Если блок ещё не создан, то создаём:
        if not Assigned(Block) then Block:= TPairsBlock.Create;
        Block.Insert(Pair);  // вставляем пару в блок
      end;
      Link:= Node.InLinkNext;  // следующий линк
    end; // while

    // Если блок создан, вставляем в множество результата:
    if Assigned(Block) then begin
      Block.mLabel:= Node;   // метка блока
      Block.mCost:= Min;     // вес легчайшей пары в блоке
      Result.Insert(Block);  // вставка блока
    end;
    Node.mColor:= CBlack; // отмечаем, что вершина обработана
    Node:= GetBestNode;   // выбор самой "мощной" на текущий момент вершины
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Рекурсивная процедура обработки блока

procedure BlockHandle(aBlock: integer;  // номер блока
                      aNodes: TSet;     // накопленные вершины
                      aRes: TCostSet    // накопленные пары
                      );
var Block: TPairsBlock; // текущий блок
    Nodes: TSet;        // Накопленное множество вершин
    Res  : TCostSet;    // Накопленное множество пар (рёбер)
    Pair : TPair;       // очередная пара из блока

    // Прогноз результата

    function Prediction(aBlock: TPairsBlock;  // текущий блок
                        aNodes: TSet;         // накопленные вершины
                        aRes: TCostSet        // накопленные пары
                        ): boolean;
    var B: TPairsBlock;         // очередной блок
        Cost, Count : integer;  // прогнозируемые цена и мощность п/с
    begin
      Result:= not Assigned(aRes);
      if Result then Exit;
      // Начинаем накопление с текущей стоимости и мощности
      Cost:= aRes.mCost;  Count:= 0;
      if Assigned(aRes.mSet) then Count:= aRes.mSet.GetCount;
      // Пропускаем блоки вплоть до текущего
      B:= Blocks.GetFirst as TPairsBlock;
      while B <> aBlock do B:= Blocks.GetNext as TPairsBlock;
      // Переход к следующему блоку после текущего
      B:= Blocks.GetNext as TPairsBlock;
      // Перебор оставшихся блоков
      while Assigned(B) do begin
        if not aNodes.Exist(B.mLabel) then begin
          Inc(Count);         // накопление мощности паросочетания
          Inc(Cost, B.mCost); // накопление прогнозируемой стоимости
        end;
        B:= Blocks.GetNext as TPairsBlock;
      end;
      // Формируем положительный результат,
      // если выполняется одно из условий:
      // - есть шанс увеличить мощность паросочетания, или
```

```
      // - при той же мощности уменьшить стоимость паросочетания
      Result:= (Count > BestCnt) or
               (Count = BestCnt) and (Cost < BestCost);
    end;

begin  { BlockHandle }
  // Извлекаем текущий блок:
  Block:= Blocks.GetItem(aBlock) as TPairsBlock;
  // Если метка текущего блока НЕ содержится в накопителе вершин,
  // то перебираем пары текущего блока
  if not aNodes.Exist(Block.mLabel) then begin
    // Вершина, которою помечен блок, ещё не входит в паросочетание
    // Перебор пар текущего блока:
    Pair:= Block.GetFirst as TPair;
    while Assigned(Pair) do begin
      // Попытка прилепить очередную пару (ребро)
      // Пара присоединяется, если множество пары
      // не пересекается с текущим множ. чёрных вершин
      if  not aNodes.TestIntersect(Pair.mSet) then begin
        // Здесь пару можно присоединить
        Nodes:= aNodes.Copy as TSet;     // копия накопленных вершин
        Res:= aRes.Copy as TCostSet;     // копия накопленных пар
        // Пристраиваем к множеству пар очередную пару
        Nodes.Add(Pair.mSet);   // накапливаем множество вершин
        Res.Insert(Pair);       // накапливаем множество пар
        if aBlock < Blocks.GetCount then begin
          // Это не последний блок.
          // Если обработка последующих блоков может увеличить
          // мощность паросочетания или снизить цену,
          // то войти в следующий блок
          if Prediction(Block,Nodes,Res)
            then BlockHandle(aBlock+1,Nodes,Res);
        end else begin
          // Достигнут последний блок (aBlock = Blocks.GetCount),
          // Запоминаем результат, если выполняется одно из условий:
          // - текущий результат Result ещё не определён
          // - мощность нового п/с превышает мощность текущего
          // - мощности п/с равны, но стоимость нового меньше
          if not Assigned(Result) or
             (Res.mSet.GetCount > BestCnt) or
             (Res.mSet.GetCount = BestCnt) and (BestCost > Res.mCost)
            then begin
              BestCost:= Res.mCost;         // лучшая (меньшая) стоимость
              BestCnt:= Res.mSet.GetCount; // текущая лучшая мощность п/с
              Result.Free;                  // удаляем прежнее паросочетание
              Result:= Res.Copy as TCostSet;// и сохраняем новое
            end;
        end;
        // Перед выбором следующей пары в блоке освобождаем текущие:
        Res.Free;   // накопленные пары
        Nodes.Free; // накопленные вершины
      end; // if
      Pair:= Block.GetNext as TPair;  // следующая пара в блоке
    end; // while
  end; // if
  // Если ищется наибольшее паросочетание без учёта веса,
  // и оно найдено, то выход
  if (aMode = pMaxN) and (BestCnt = Self.Nodes div 2) then Exit;
  // После перебора блока входим в следующий, если он существует
  if aBlock < Blocks.GetCount then begin
    // Текущий блок не последний.
    // Если обработка последующих блоков может увеличить
```

```
      // мощность паросочетания или снизить цену, то войти в следующий блок
      if Prediction(Block,aNodes,aRes) then
         BlockHandle(aBlock+1,aNodes,aRes);
    end else begin
      // Достигнут последний блок (aBlock = Blocks.GetCount),
      // Запоминаем результат, если выполняется одно из условий:
      // - текущий результат Result ещё не определён
      // - мощность нового п/с превышает мощность текущего
      // - мощности п/с равны, но стоимость нового меньше
      if not Assigned(Result) or
         (aRes.mSet.GetCount > BestCnt) or
         (aRes.mSet.GetCount = BestCnt) and (BestCost > aRes.mCost)
        then begin
          BestCost:= aRes.mCost;          // лучшая (меньшая) стоимость
          BestCnt:= aRes.mSet.GetCount;   // текущая лучшая мощность п/с
          Result.Free;                    // удаляем прежнее паросочетание
          Result:= aRes.Copy as TCostSet;// и сохраняем новое
        end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Инверсия знаков весов рёбер паросочетания
  // (используется при поиске паросочетания максимального веса)

  procedure RestoreCost;
  var Pair : TPair;
  begin
    // Перебираем множество пар паросочетания Result.mSet
    with Result.mSet do begin
      Pair:= GetFirst as TPair;
      while Assigned(Pair) do begin
        Pair.mCost:= -Pair.mCost;
        Pair:= GetNext as TPair;
      end;
    end;
    Result.mCost:= -Result.mCost;  // итоговая стоимость всех пар п/с
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Перед уничтожением блоков удаляем из них найденные паросочетания

  procedure RemoveFromBlocks;
  var  Block: TPairsBlock;
  begin
    with Blocks do begin
      Block:= GetFirst as TPairsBlock;
      while Assigned(Block) do begin
        Block.Sub(Result.mSet);
        Block:= GetNext as TPairsBlock;
      end;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenPairs }
  Result:= nil;        // Результат -- лучшее паросочетание
  Blocks:= GenBlocks;  // Создаём блоки
  if Blocks.GetCount = 0 then begin Blocks.Free; Exit; end;

  // Подготовка переменных:
  BestCost:= MaxInt;            // текущая лучшая сумма
  BestCnt:= 0;                  // текущая лучшая (наибольшая) мощность п/с
  Nodes:= CreateSet;            // множество вершин паросочетания
```

```
  Pairs:= TCostSet.CreateEmpty; // начальное паросочетание

  // Рекурсивная обработка блоков:
  BlockHandle(1, Nodes, Pairs);  // формирует Result

  // Очистка памяти:
  Pairs.Free;         // Начальное паросочетание (пустое)
  Nodes.Free;         // Начальное множество присоединённых вершин
  // Удаляем из блоков множество Result.mSet
  if Assigned(Result) then  RemoveFromBlocks;
  Blocks.ClrAndDestroy;  // Удаляем сами блоки
  Blocks.Free;           // и множество блоков

  // После поиска паросочетания максимальной стоимости
  // восстанавливаем веса рёбер:
  if aMode = pMaxW then RestoreCost;
end;


//#########################################################################
//                                                                       #
//               Паросочетания в двудольном графе                        #
//                                                                       #
//#########################################################################

// Построение минимального совершенного паросочетания.
// Алгоритм был предложен Кёнигом, Эгервари и Куном (венгерский алгоритм)
// Кристофидес, стр. 405
// Обрабатываются только активные дуги, для которых mHigh=0
// Возвращает стоимость паросочетания и его мощность aPairs

// Поля объектов используются следующим образом:
// * TNode.mColor -- белая - экспонированная, чёрная - в паре
// * TNode.mDist  -- вершинное число, степень "поднятия мостовой опоры"
// * TNode.mFlow  -- признак посещения вершины (0/1)
// * TNode.mRoot  -- линк, ведущий к корню текущего дерева
// * TNode.mLink  -- чёрный линк паросочетания для этой вершины
//
// * TLink.mColor -- чёрная - признак принадлежности к паросочетанию
// * TLink.mValue -- стоимость (длина, крутизна) дуги
// * TLink.mHigh  -- признак активности дуги, 0 - активна, 1 - отключена
// * TLink.mLow   -- редуцированная (пониженная) стоимость дуги

function TGraph.MarkMinPairsDicoty(var aPairs: integer): integer;

var    Que: TBuffer;  // очередь вершин
    NodesL: TSet;     // множество левых вершин
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Выполняется начальная установка двудольного графа:
  // все вершины и дуги белые
  // Формируется вспомогательное подмножество левых вершин
  // Возвращает максимально возможное количество пар

  function  InitNodes: integer;
  var Node: TNode;
      Link: TLink;
      L, R: integer; // счётчики левых и правых вершин
  begin
    // Обнуление счётчиков вершин
    L:= 0; R:= 0;
    // Перебор всех вершин:
    Node:= NodeFirst;
```

```
    while Assigned(Node) do begin
      Node.mColor:= CWhite;   // цвет вершины белый
      if Node.IsLeft then begin
        Inc(L);               // подсчёт левых вершин
        Node.mDist:= MaxInt;  // вершинное число ищем вторым перебором
        NodesL.Insert(Node);  // и вставка в множество левых
      end else if Node.IsRight then begin
        Inc(R);               // подсчёт правых вершин
        Node.mDist:=0;        // вершинное число = 0
      end;
      Node:= NodeNext;
    end; // while
    // Количество ожидаемых паросочетаний равно меньшему из двух:
    if L > R then Result:= R else Result:= L;

    // Перебор левых вершин:
    Node:= NodesL.GetFirst as TNode;
    while Assigned(Node) do begin
      // Перебор исходящих линков
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do if mHigh=0 then begin
          // это активный линк
          Link.mColor:= CWhite;
          // запоминаем вес минимальной дуги
          if Node.mDist > Link.mValue  then
            Node.mDist:= Link.mValue; // начальное "приподнятие опоры"
        end;
        Link:= Node.OutLinkNext;
      end; // while
      Node:= NodesL.GetNext as TNode;
    end; // while

  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск аугментальной (улучшающей) цепи:
  // aLeft - корневая экспонированная левая вершина дерева (белая)
  // Возвращает экспонированную правую вершину (белую),
  // при условии, что найдена аугментальная цепь, а иначе NIL.
  // Формирует ссылки в полях TNode.mRoot, ведущие к корню дерева.
  // Если экспонированная вершина не найдена,
  // то параметр aDelta содержит минимальное приращение цены,
  // необходимое для покупки ещё хотя бы одного линка.

  function FindAugmenting(aLeft : TNode; var aDelta: integer): TNode;
  var Node: TNode;
      Link: TLink;
      Cost: integer;
  begin
    Result:= nil;         // ссылка на правую вершину
    Que.Clear;            // очистить очередь вершин
    Que.Put(aLeft);       // и поместить туда корневую вершину
    // Пока очередь не пуста и не обнаружена аугментальная цепь
    while (Que.GetCount > 0) and not Assigned(Result) do begin
      Node:= Que.Get as TNode;
      Node.mFlow:= 1;    // отмечаем посещение вершины
      if Node.IsLeft then begin
        // Это вершина левой доли
        // Ищем белую вершину в правой доле, а чёрные ставим в очередь
        Link:= Node.OutLinkFirst;
        while Assigned(Link) do begin
          // обрабатываем только активные белые линки
```

```
          with Link do if (mHigh=0) and     // активный линк
                          (mColor=CWhite)   // и линк белый
            then begin
              // Вычисляем приведенную (пониженную) стоимость дуги
              Cost:= mValue - (mOwner.mDist - mDest.mDist);
              if Cost = 0 then begin
                // Это достижимая правая вершина (Cost=0)
                if  mDest.mColor = CWhite then begin
                  // Если она белая,
                  // то найдена белая вершина и аугментальная цепь
                  Result:= mDest;       // экспонированная белая вершина
                  Result.mRoot:= Link; // ссылка в направлении корня дерева
                  Break;               // выход из цикла и процедуры
                end else if mDest.mFlow = 0 then begin
                  // Найдена чёрная вершина,
                  // она ещё не посещалась, и потому ставим её в очередь
                  mDest.mRoot:= Link; // ссылка в направлении корня дерева
                  Que.Put(mDest);     // ставим в очередь
                end; // else
              end else begin
                // Это недостижимая правая вершина (Cost > 0),
                // через aDelta возвращаем минимальное приращение цены
                if aDelta > Cost then aDelta:= Cost;
              end;
            end; // with Link
            Link:= Node.OutLinkNext;
          end; // while - конец перебора линков
        end else begin
          // Это внутренняя чёрная вершина в правой доле (Node.mIsLeft = FALSE)
          Link:= Node.mLink;         // извлекаем из неё линк паросочетания
          Link.mOwner.mRoot:= Link; // это ссылка в направлении корня дерева
          // Ставим в очередь найденную внешнюю чёрную вершину в левой доле
          Que.Put(Link.mOwner);
        end;
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Инверсия аугментальной (увеличивающей) цепи.
    // Вызывается при условии, что такая цепь обнаружена
    // в процедуре FindAugmenting.
    // aLeft, aRight -- левая и правая экспонированные (белые) вершины.
    // Использует поля Node.mRoot, сформированные процедурой FindAugmenting,
    // и формирует из них линки паросочетаний Node.mLink.

    procedure Inverse(aLeft, aRight : TNode);
    var Node: TNode;
        Link: TLink;
        Flag: boolean;
    begin
      // Проходим аугментальную цепь в обратном порядке
      // инвертируя цвета дуг
      Flag:= true;      // инвертируемый флаг определяет цвет ребра (дуги)
      Node:= aRight;
      repeat
        Link:= Node.mRoot as TLink;
        if Flag then begin
          // Переход справа налево, окраска дуги чёрным
          // Красим белым старый линк паросочетания
          with Node do if Assigned(mLink) then mLink.mColor:= CWhite;
          Node.mLink:= Link;       // запоминаем новый линк паросочетания
          Link.mColor:= CBlack;    // и красим его чёрным
          Node:= Link.mOwner;      // переход в сторону корня дерева
```

```
      Node.mLink:= Link;      // здесь запоминаем новый линк паросочетания
    end else begin
      // Переход слева направо, окраска дуги белым
      Link.mColor:= CWhite;
      Node:= Link.mDest;     // переход в сторону корня дерева
    end;
    Flag:= not Flag; // инвертируемый флаг определяет цвет ребра (дуги)
  until Node = aLeft;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Очистка у всех вершин полей TNode.mFlow -- признаков посещения.
// Вызывается перед обходом свободных (белых) левых вершин

procedure ClearFlags;
var Node: TNode;
begin
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mFlow:= 0;   // Это признак посещения при постройке деревьев
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Обход свободных (белых, экспонированных) левых вершин
// с целью построения улучшающих аугментальных цепей.
// Возвращает количество обнаруженных цепей (и соответственно пар).
// Если ни одна такая цепь не будет обнаружена,
// то через параметр aDelta возвращает минимальное "поднятие опоры",
// необходимое для выпрямления хотя бы одного моста

function FindPairs(var aDelta: integer): integer;
var LNode, RNode : TNode;
begin
  Result:= 0;        // счётчик обнаруженных пар
  aDelta:= MaxInt;   // минимальное приращение цены
  ClearFlags;        // очистка признаков посещения вершин TNode.mFlow
  // Перебор левых вершин
  LNode:= NodesL.GetFirst as TNode;
  while Assigned(LNode) do begin
    if LNode.mColor = CWhite then begin
      // Это открытая (экспонированная) левая вершина
      // Ищем открытую вершину справа и формируем aDelta
      RNode:= FindAugmenting(LNode, aDelta);
      if Assigned(RNode) then begin
        // Аугментальная цепь найдена
        // Инвертируем цвета дуг от RNode к LNode
        Inverse(LNode, RNode);
        // Две крайние вершины цепи метим чёрным
        // как вошедшие в паросочетание
        LNode.mColor:= CBlack;
        RNode.mColor:= CBlack;
        Inc(Result);  // и наращиваем счётчик пар
      end;
    end;
    LNode:= NodesL.GetNext as TNode;
  end;
end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура корректирует вершинные числа Node.mDist на величину aDelta
  // (приподнимает платформу)
  // Вызывается после обхода экспонированных (белых) вершин
  // после того, как ни одна новая пара не обнаружена.

  procedure Correct(aDelta: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Если вершина посещалась, то нарастить вершинное число
      if Node.mFlow <> 0 then begin
        Inc(Node.mDist, aDelta);
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт стоимости паросочетания по сумме чёрных линков
  // и формирование пониженной стоимости (крутизны) дуг TLink.mLow
  // с учётом текущих значений вершинных чисел TNode.mDist.
  // (поля TLink.mLow используются для поиска контура Гамильтона)

  function CalcCost: integer;
  var Node: TNode;
      Link: TLink;
  begin
    Result:= 0;
    // Перебор вершин левой доли
    Node:= NodesL.GetFirst as TNode;
    while Assigned(Node) do begin
      // Node.mLink - это чёрный линк паросочетания
      Inc(Result, Node.mLink.mValue);
      // Перебор исходящих линков
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // если это активный линк, вычисляем редуцированную стоимость
        with Link do
          if mHigh=0 then mLow:= mValue - (mOwner.mDist - mDest.mDist);
        Link:= Node.OutLinkNext;
      end;
      Node:= NodesL.GetNext as TNode;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
   P: integer;     // количество новых пар на очередной итерации
   Delta: integer; // приращение, необходимое для приподнятия платформы
   Count: integer; // счётчик осташихся не найденных пар

begin  { TGraph.SetPairsDicoty }

  Que:= TBuffer.Create; // очередь вершин
  NodesL:= CreateSet;   // множество левых вершин
  Count:= InitNodes; // возвращает максимально возможное число пар
  aPairs:= Count;

  repeat     // повторять до формирования всех паросочетаний
    repeat   // повторять, пока возможно приращение паросочетания
      P:= FindPairs(Delta); // P - количество обнаруженных новых пар
                            // Delta - миним. приращение для покупки дуги
```

```
      Dec(Count, P);         // Count - счётчик оставшихся пар
    until (P=0) or (Count=0);
    // Если паросочетание не полно, то скорректировать вершинные числа
    // на величину Delta (приподнять платформу),
    // "выровняв" тем очередной мост или несколько мостов
    if Count > 0 then Correct(Delta);
  until Count=0;
  // Здесь максимальное паросочетание достигнуто,
  // подсчитать стоимость паросочетания и остатки стоимостей дуг
  Result:= CalcCost;
  // Очистка памяти:
  NodesL.Free;  // множество левых вершин
  Que.Free;     // очередь вершин
end;


// Формирует множество паросочетания после того, как помечены его линки
// методом MarkMinPairsDicoty

function TGraph.GenMinPairsDicoty: TCostSet;
var Cost: integer;   // стоимость
    Pairs: integer;  // количество найденных пар
    Node: TNode;
    S : TSet;
begin
  Cost:= MarkMinPairsDicoty(Pairs);
  S:= CreateSet;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft
      then S.Insert(Node.mLink);
    Node:= NodeNext;
  end;
  Result:= TCostSet.Create(Cost, S, true);
end;


// Построение наибольшего паросочетания минимальной стоимости
// через вычисление ПОТОКА минимальной стоимости

function TGraph.GenMinPairsDicotyFlow: TCostSet;
var  S, T : TNode;
     Node : TNode;
     Link : TLink;
     Cost : integer;
      Cnt : integer;
      Res : TSet;
begin
  // Всем дугам графа назначаем единичную пропускную способность
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      with Link do begin
        mTemp:= mHigh;  // временно сохраняем
        mHigh:= 1;      // и устанавливаем единичную пропускную способность
      end;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
```

```
  // Создаём искусственные исток S и сток R:
  S:= TNode.Create(0, Self);
  T:= TNode.Create(0, Self);

  // Перебираем вершины, соединяя левую долю с истоком, а правую со стоком
  // дугами со стоимостью =0, пропускной способностью =1
  // Одновременно подсчитываем вершины в левой доле
  Cnt:= 0;
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.IsLeft then begin
      // Дуга из истока в левую долю,
      Link:= S.MakeLink(Node, 0);
      Link.mHigh:= 1;
      Inc(Cnt);
    end  else begin
      // Дуга из правой доли в сток,
      Link:= Node.MakeLink(T, 0);
      Link.mHigh:= 1;
    end;
    Node:= NodeNext;
  end;  // while

  // Из количества вершин в левой и правой долях выбираем меньшее:
  if Cnt > mNodes.GetCount div 2 then Cnt:= mNodes.GetCount - Cnt;

  // Вставляем в граф:
  InsertNode(S);    // исток
  InsertNode(T);    // сток

  // Находим поток минимальной стоимости:
  Cost:= CalcMinCostFlow(S, T, Cnt);

  // Удаляем и уничтожаем искусственные исток и сток:

  RemoveNode(T);
  RemoveNode(S);
  T.Free; S.Free;

  Res:= CreateSet;
  if Cost>=0 then begin
    // Здесь паросочетание существует
    // В паросочетание включаем дуги с ненулевым потоком (Link.mFlow <> 0)
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        with Link do begin
          if mFlow <> 0 then Res.Insert(Link);
          mHigh:= mTemp; // восстанавливаем исходные значения
        end;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
  end; // if
  // Формируем множество со стоимостью:
  Result:= TCostSet.Create(Cost, Res, true);
end;
```

```
//#########################################################################
//                                                                        #
//             Цикл Эйлера и Задача почтальона                            #
//                                                                        #
//#########################################################################

// В зависимости от того, ориентирован граф, или нет,
// вызывает метод для орграфа или неориентированного графа

function TGraph.GenPostPath(var aCost: integer): TBuffer;
begin
  if mDirect
    then Result:= GenPostPathDir(aCost)
    else Result:= GenPostPathUndir(aCost)
end;

// Генерация цикла (цепочки рёбер) в неориентированном графе

function TGraph.GenPostPathUndir(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Изменение степени ребра (двух полеё TLink.mLimit)

  procedure ModifyLimit(aLink: TLink; aDelta: integer);
  var L: TLink; // встречный линк
  begin
    Inc(aLink.mLimit, aDelta);               // прямой линк
    L:= aLink.mDest.GetLink(aLink.mOwner);   // встречный линк
    Inc(L.mLimit, aDelta);
  end;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Формирование множества вершин с нечётной степенью
  // и предварительная подготовка полей TNode.mLimit

  function GenOddNodes: TSet;
  var Node: TNode;
  begin
    Result:= nil;
    // Перебор всех вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      // Сброс:  mColor= 0;  mPred= mLink= nil;  mDist= mFlow= MaxInt;
      Node.ResetNode;
      // В поле mLimit сохраним количество линков вершины (степень):
      Node.mLimit:= Node.OutGetCnt;  //Node.mLnkOut.GetCount;
      // Если количество рёбер нечётное, заносим в множество результата:
      if Odd(Node.mLimit) then begin
        if not Assigned(Result) then Result:= CreateSet;
        Result.Insert(Node);
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Установка степеней рёбер в полях TLink.mLimit
  // и степеней вершин в полях TNode.mLimit

  procedure LimitsPrepare;
  var OddNodes: TSet;     // множество вершин с нечётной степенью
        Pairs : TCostSet; // паросочетание минимального веса
         Pair : TPair;    // пара вершин
```

```
          Gr : TGraph;    // вспомогательный граф
    Ni, Nj   : TNode;     // вершины исходного графа
    N2i,N2j  : TNode;     // копии вершин исходного графа во вспомогательном
       NFin  : TNode;     // конечная вершина в кратчайшем пути
       Dist  : integer;   // кратчайшее расстояние между вершинами
       Link  : TLink;
       i, j  : integer;
begin
  // Устанавливаем степени всех рёбер в единицу:
  SetLinksLimit(1);
  // Собираем множество вершин с нечётными степенями:
  OddNodes:= GenOddNodes;
  // Если таких вершин нет, то выход:
  if not Assigned(OddNodes) then Exit;

  // Создаём вспомогательный граф:
  Gr:= TGraph.Create('Odd Nodes:', // имя графа произвольно
                     false,        // граф не ориентирован
                     false,        // вершины не нагружены
                     true          // рёбра нагружены
                    );
  // И вставляем в него копии вершин с нечётными степенями:
  Ni:= OddNodes.GetFirst as TNode;
  while Assigned(Ni) do begin
    Gr.InsertNode(Ni.Copy as TNode);
    Ni:= OddNodes.GetNext as TNode;
  end;
  // Для определения взаимных расстояний строим карту дальних указателей
  // в исходном графе:
  Self.InitMap_Floyd;
  // Строим рёбра между всеми вершинами вспомогательного графа
  // (создаём полный граф).
  // Длина рёбер устанавливается равной длине кратчайшего пути
  // между вершинами исходного графа
  for i:= 1 to OddNodes.GetCount-1 do begin
    Ni:= OddNodes.GetItem(i) as TNode;     // вершина из множества нечётных
    N2i:= Gr.mNodes.GetItem(i) as TNode;   // её копия в графе
    // Перебор последующих после Ni вершин:
    for j:= i+1 to OddNodes.GetCount do begin
      Nj:= OddNodes.GetItem(j) as TNode;   // вершина из множества нечётных
      N2j:= Gr.mNodes.GetItem(j) as TNode; // её копия во вспомогат. графе
      Dist:= Ni.GetFarLink(Nj).mDist; // кратчайшее расстояние в исх. графе
      Gr.SetLink(N2i, N2j, Dist);     // назначим длине ребра во вспомогат.
    end;
  end;
  // Во вспомогательном графе
  // находим паросочетание с минимальным весом:
  Pairs:= Gr.GenPairs(pMinW);

  // Для каждой из полученных пар увеличиваем степени mLimit
  // вдоль соответствующих кратчайших путей:
  Pair:= Pairs.mSet.GetFirst as TPair;
  while Assigned(Pair) do begin
    // Pair.mSet содержит ровно две вершины пары.
    // Стартовая вершина цепи в исходном графе:
    Ni:= mNodes.GetObject(Pair.mSet.GetFirst as TNode) as TNode;
    // Конечная вершина цепи в исходном графе:
    NFin:= mNodes.GetObject(Pair.mSet.GetNext as TNode) as TNode;
    repeat
      Inc(Ni.mLimit);          // степень+1 в начальной (промежут.) вершине
      Nj:= Ni.GetNear(NFin);   // следующая вершина на кратчайшем пути
      Link:= Ni.GetLink(Nj);   // линк на промежуточную вершину
```

```
        ModifyLimit(Link, +1);  // степень+1 линка
        Ni:= Nj;                // продвижение к следующей вершине
      until Ni = NFin;          // пока не достигнута конечная
      Inc(Ni.mLimit);           // степень+1 в конечной вершине
      Pair:= Pairs.mSet.GetNext as TPair;  // следующая пара
    end;
    // Очистка памяти:
    Pairs.ClrAndDestroy;     // пары
    Pairs.Free;              // множество пар
    DoneMap;                 // освобождаем карту
    Gr.Free;                 // вспомогательный граф
    OddNodes.Free;           // множество вершин с нечётной степенью
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление кольцевых маршрутов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк
      // (степень которого Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл и путь:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (степень больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины пути)
      ModifyLimit(Link, -1);    // уменьшаем степень ребра
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;
    Link: TLink;
    Count: integer; // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathUndir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= 0;                // накопитель стоимости
  LimitsPrepare;            // подготовка полей mLimit (пределов посещ.)
  Node:= NodeFirst;         // построение начинаем с первой вершины

  // Строим кольцевые маршруты,
  // пока не исчерпаны степени всех вершин и рёбер
  repeat
    // Строим кольцевой маршрут из вершины Node и добавляем в результат:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;        // счётчик для прокрутки буфера Result
    while Count>0 do begin
      Link:= Result.Get as TLink;   // берём линк из начала буфера
      Result.Put(Link);             // и помещаем в конец
```

```
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то стоп.
      // Построение очередного контура начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
    // Выход из цикла, когда степени всех вершин и рёбер исчерпаны:
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;


//   Задача почтальона для сильно связанного орграфа

function TGraph.GenPostPathDir(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подготовка полей mLimit в дугах и вершинах графа.
  // Возвращает стоимость дополнительного потока

  function LimitsPrepare: integer;
  var Node : TNode;    // текущая вершина
      Link : TLink;    // текущая дуга
      Flag : boolean;  // признак наличия асимметричных вершин
      S, T : TNode;    // искусственные исток и сток
      Flow: integer;   // величина корректирующего потока
  begin
    Result:= 0;
    // Устанавливаем пределы посещения всех дуг в единицу:
    SetLinksLimit(1);
    // Для всех вершин определяем разности полустепеней входа и выхода,
    // а также степени вершин TNode.mLimit:
    Flag:= false;        // признак асимметрии графа
    // Перебор вершин:
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mLimit:= InGetCnt + OutGetCnt;  // степень вершины
        mDist:= InGetCnt - OutGetCnt;   // асимметрия
        Flag:= Flag or (mDist<>0); // признак наличия асимметричных вершин
      end;
      Node:= NodeNext;
    end;
    // Если все вершины симметричны (все полустепени совпадают),
    // то выход из процедуры:

    if not Flag then Exit;
```

```
  // Для несимметричного графа строим поток минимальной стоимости
  // с величиной, равной суммарной ассиметрии
  // Создаём:
  S:= TNode.Create(0, Self); // вспомогательный исток
  T:= TNode.Create(0, Self); // вспомогательный сток

  Flow:= 0; // здесь подсчитаем суммарный поток
  // Обработка всех вершин, кроме S и T:
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Во всех дугах формируем данные для вычисления потока:
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Link.mLow:= 0;            // минимальный поток = 0
      Link.mHigh:= MaxInt;      // максимальный не ограничен
      Link:= Node.OutLinkNext;
    end;
    // Если вершина несимметрична (mDist <> 0),
    // соединяем её с искусственным истоком либо стоком:
    if Node.mDist < 0 then begin
      // создаём дугу из вершины во вспомогательный сток:
      Link:= Node.MakeLink(T, 0); // создаём линк
      Link.mHigh:= -Node.mDist;   // пропускная способность дуги
    end else if Node.mDist > 0 then begin
      // Накапливаем требуемый поток:
      Inc(Flow, Node.mDist);      // асимметрия вершины
      // создаём дугу из вспомогательного истока в вершину:
      Link:= S.MakeLink(Node, 0); // создаём линк
      Link.mHigh:= Node.mDist;    // пропускная способность дуги
    end;
    Node:= NodeNext;
  end; // while
  // Вставляем вспомогательные вершины в граф:
  mNodes.Insert(S);         // исток
  mNodes.Insert(T);         // сток

  // Распределяем поток Flow по дугам и подсчитываем его стоимость:
  Result:= CalcMinCostFlow(S, T, Flow);

  // Удаляем из графа и освобождаем:
  RemoveNode(S);  S.Free;     // вспомогательный исток
  RemoveNode(T);  T.Free;     // вспомогательный сток

  // Добавляем полученные потоки в дугах к степеням дуг и вершин
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Link:= Node.OutLinkFirst;
    while Assigned(Link) do begin
      Flow:= Link.mFlow;  // поток в дуге
      // При наличии потока
      // добавляем его к степеням дуги и двух смежных вершин:
      if Flow > 0 then begin
        Inc(Link.mLimit, Flow);          // + к дуге
        Inc(Link.mOwner.mLimit, Flow);   // + к источнику дуги
        Inc(Link.mDest.mLimit, Flow);    // + к приёмнику дуги
      end;
      Link:= Node.OutLinkNext;
    end;
    Node:= NodeNext;
  end;
end;
```

```
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Добавление частичных циклов из вершины aNode

  procedure AddPath(aNode: TNode);
  var  Link: TLink;
  begin
    while true do begin
      // Ищем первый попавшийся открытый линк (Link.mLimit > 0)
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) and (Link.mLimit <= 0)
        do Link:= aNode.OutLinkNext;
      // Если нет открытых линков, то завершаем цикл:
      if not Assigned(Link) then Break;
      // Здесь линк открыт (лимит больше нуля):
      Result.Put(Link);         // помещаем линк в буфер результата
      Dec(aNode.mLimit);        // уменьшаем степень текущей вершины
      aNode:= Link.mDest;       // продвигаемся к следующей вершине
      Dec(aNode.mLimit);        // уменьшаем степень следующей вершины
      Inc(aCost, Link.mValue); // накопление стоимости (длины)
      Dec(Link.mLimit);         // уменьшаем степень дуги
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var Node: TNode;      // текущая вершина
    Link: TLink;      // текущая дуга
    Count: integer;   // счётчик для прокрутки буфера Result

begin { TGraph.GenPostPathDir }

  Result:= TBuffer.Create;  // пустой буфер для результата
  aCost:= -1;     // Стоимость цепи на случай отсутствия решения

  // Если граф не сильно связан, то решения нет:

  if not IsLinked then Exit;

  aCost:= 0;           // накопитель стоимости
  LimitsPrepare;       // подготовка степеней mLimit вершин и рёбер
  Node:= NodeFirst;    // построение начинаем с любой вершины

  // Цикл добавления контуров к буферу Result:
  repeat
    // Строим циклы из текущей вершины Node
    // и добавляем в буфер Result:
    AddPath(Node);

    // Прокручиваем текущий буфер (маршрут) в поисках вершины
    // с ненулевой степенью (Node.mLimit > 0)
    Node:= nil;
    Count:= Result.GetCount;         // счётчик для прокрутки буфера
    while Count>0 do begin
      Link:= Result.Get as TLink;    // берём линк из начала буфера
      Result.Put(Link);              // и помещаем в конец
      Dec(Count);
      // Если степень mLimit следующей вершины не нулевая, то
      // построение очередного контура начинаем с вершины Link.mDest
      if Link.mDest.mLimit > 0 then begin
        Node:= Link.mDest;
        Break;
      end;
    end;
```

```
    // Выход, если степени всех вершин исчерпаны
  until not Assigned(Node);

  // Докручиваем буфер так, чтобы первая вершина стала в начало:
  Node:= NodeFirst;
  Count:= Result.GetCount;
  while Count>0 do begin
    Link:= Result.Get as TLink; // берём линк из начала буфера
    // Это линк из первой вершины?
    if Link.mOwner = Node then begin
      Result.Push(Link);        // да, возвращаем назад в буфер
      Break;                    // и прекращаем цикл
    end;
    Result.Put(Link);           // иначе помещаем в конец буфера
    Dec(Count);
  end;
end;

//############################################################################
//                                                                           #
//            Задача коммивояжёра и контур Гамильтона                        #
//                  Майника, гл. 7, стр. 251                                 #
//                                                                           #
//############################################################################

//  Поиск контура Гамильтона полным перебором в глубину

function TGraph.GenHamilton_Full(var aCost: integer): TBuffer;
var
    Cost: integer;    // текущая стоимость контура
    Count: integer;   // счётчик окрашенных вершин
    First: TNode;     // стартовая вершина
    Node: TNode;      // текущая вершина
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aNode - последняя окрашенная вершина
  procedure SaveResult(aNode: TNode);
  begin
    aCost:= Cost;          // сохраняем стоимость контура
    First.mPred:= aNode;   // замыкаем контур
    Result.Clear;          // очищаем буфер результата
    Node:= aNode;          // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=First;      // пока не достигнем исходной вершины
    Result.Push(Node);     // заносим исходную вершину
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);
  var L : TLink;  // текущий исходящий линк
  begin
    aNode.mColor:= CBlack;  // красим данную вершину
    Inc(Count);             // и наращиваем счётчик окрашенных
    // Все вершины окрашены?
    if Count < mNodes.GetCount then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
```

```
        if L.mDest.mColor = CWhite then begin
          // Нашли неокрашенную:
          Inc(Cost, L.mValue);         // накапливаем стоимость
          if Cost < aCost then begin   // если стоимость ниже минимальной
            L.mDest.mPred:= aNode;     // то метим предыдущей вершиной
            Local(L.mDest);            // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);         // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;         // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount -- все вершины окрашены
      // Ищем замыкающий линк на стратовую вершину First
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>First)
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);       // накапливаем стоимость
        if Cost < aCost            // если она меньше минимальной
          then SaveResult(aNode); // сохраняем контур и его стоимость
        Dec(Cost, L.mValue);       // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenHamilton_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                  // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  aCost:= MaxInt;// текущая минимальная стоимость = Maxint
  ResetNodes;           // очистка вспомогательных полей
  Cost:= 0;             // накопленная стоимость = 0
  Count:= 0;            // счётчик окрашенных вершин = 0
  First:= NodeFirst;    // стартовая вершина
  Local(First);         // вызов рекурсивной процедуры
  // Если контур не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;


//  Упрощённый жадный алгоритм
//  поиска минимального Гамильтонова контура

function TGraph.GenHamilton_Greed(var aCost: integer;
                                  aGreed: boolean
                                  ): TBuffer;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
         (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
```

```
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Start: TNode;     // начальная вершина
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamilton_Greed }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                 // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;           // очистка вспомогательных полей
  Start:= NodeFirst;    // начинаем с первой вершины
  Node:= Start;
  Cost:= 0;             // накопленная стоимость = 0
  Count:= mNodes.GetCount-1; // счётчик неокрашенных вершин
  while Count > 0 do begin
    Node.mColor:= CBlack;    // красим очередную вершину
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);    // накапливаем стоимость
    Dec(Count);             // счётчик - 1
    Node:= L.mDest;         // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины
    Result.Put(Node);        // заносим в результат предпоследнюю
    L:= Node.GetLink(Start); // линк из последней в первую
    if Assigned(L) then begin
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else begin
      Result.Clear;
```

```
    end;
  end;
end;

//        Поиск минимального Гамильтонова контура
// Усовершенствованный алгоритм на базе двудольного графа


type  // Специальная вершина для вспомогательного двудольного графа

      THamNode = class (TNode)
        mLeft : boolean;     // признак левой вершины (TRUE)
        mNode : TNode;       // родовая вершина
        mBack : THamNode;    // ссылка справа налево и слева направо
        constructor Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
        function  Compare(arg: TItem): TCompare; override;
        function  GetName: string;  override;
        function  NextCircNode: THamNode;
        function  GenFactorsLinks: TSet;
      end;

constructor THamNode.Create(aOwner: TGraph; aNode: TNode; aLeft: boolean);
begin
  inherited Create(0, aOwner);
  mNode:= aNode;     // родовая вершина
  mLeft:= aLeft;     // сторона: TRUE - Left, FALSE - Right
end;

function  THamNode.GetName: string;
begin
  // Сторона + имя родовой вершины
  if mLeft then Result:= 'L.' else Result:= 'R.';
  Result:= Result + mNode.GetName;
end;

// Генерирует множество неотключенных дуг,
// ведущих внутрь текущего фактора (контура)

function THamNode.GenFactorsLinks: TSet;
var Link: TLink;
begin
  Result:= CreateSet;
  Link:= OutLinkFirst;
  while Assigned(Link) do begin
    // Если дуга ведёт внутрь текущего контура (фактора)
    // и она не отключена (mHigh = 0), то сохраняем в результате
    with Link do
    if ((mDest as THamNode).mBack.mValue = Self.mValue) and (mHigh = 0)
      then Result.Insert(Link);  // запомнить в множестве отключенных
    Link:= OutLinkNext;
  end;
end;

// Возвращает следующую левую вершину текущего контура
// в соответствии с текущим потоком

function THamNode.NextCircNode: THamNode;
begin
  Result:= (mLink.mDest as THamNode).mBack;
end;
```

```
// При вставке вершины сортируются по алфавиту

function  THamNode.Compare(arg: TItem): TCompare;
begin
  Result:= cmpEq;
  if arg = Self then Exit;
  if GetName < (arg as THamNode).GetName
    then Result:= cmpLess
  else if GetName > (arg as THamNode).GetName
    then Result:= cmpGreate
end;


// Специальный двудольный граф для поиска гамильтоновых контуров
// Kristofides - 404

type  THamGraph = class (TGraph)
        mGraph: TGraph;   // исходный (родовой) граф
        constructor Create(aGraph : TGraph);
      protected
        function Check: boolean;
        function CalcFactors(var aStart: THamNode): integer;
        procedure SaveBest(aBuf: TBuffer);
        function CalcCostAndFactors(var aFactors: integer; //кол-во факторов
                                    var aStart: THamNode  // стартовая вершина
                                    ): integer;           // стоимость потока
        function Shrink(aFact: integer): THugeGraph;
      end;


constructor THamGraph.Create(aGraph : TGraph);
var Node: TNode;
    HamL, HamR: THamNode;
    Link : TLink;
begin
  inherited Create('Hamilton',  // имя
                   true,        // дуги ориентированы
                   false,       // вершины не нагружены
                   true);       // дуги нагружены

  mGraph:= aGraph;            // ссылка на исходный граф

  // На базе вершин исходного графа создаём пары специальных вершин:
  Node:= aGraph.NodeFirst;
  while Assigned(Node) do begin
    // Создаём пару вершин
    HamL:= THamNode.Create(Self, Node, True);  // исток
    HamR:= THamNode.Create(Self, Node, False); // сток
    // Формируем взаимные ссылки для ускорения доступа:
    HamL.mBack:= HamR;
    HamR.mBack:= HamL;
    // и вставляем вершины в этот граф
    mNodes.Insert(HamL);
    mNodes.Insert(HamR);
    Node:= aGraph.NodeNext;
  end;

  // Формируем дуги, ведущие из левой доли в правую:
  HamL:= NodeFirst as THamNode;
  while Assigned(HamL) do begin
    // Это левая вершина?
    if HamL.mLeft then  begin
```

```
      // Да, обрабатываем её
      PosPush;  // сохраним позицию перебора вершин
      // Перебираем исходящие дуги в вершине исходного графа
      Link:= HamL.mNode.OutLinkFirst;
      while Assigned(Link) do begin
        Node:= Link.mDest;  // целевая вершина исходного графа
        // Ищем соответствующую ей вершину в правой доле:
        HamR:= NodeFirst as THamNode;
        while Assigned(HamR) do begin
          with HamR do if not mLeft and (mNode = Node) then Break;
          HamR:= NodeNext  as THamNode;
        end;
        // Здесь HamR соответствует целевой вершине исходного графа
        // Создаём дугу из левой доли в правую (той же стоимости)
        HamL.MakeLink(HamR, Link.mValue);
        // Переход к следующей дуге исходного графа
        Link:= HamL.mNode.OutLinkNext;
      end; // while
      PosPop;  // восстановим позицию перебора вершин
    end; // if
    HamL:= NodeNext as THamNode;
  end; // while
end;


// Поиск и пометка в поле mValue всех факторов (контуров)
// Возвращает количество факторов.
// Принадлежность вершины к фактору определяется полем mValue

function THamGraph.CalcFactors(var aStart: THamNode): integer;
var NodesCnt: integer;  // счётчик оставшихся необработанных вершин
    Start, Left : THamNode; // стартовая и текущая вершины
    CircLen: integer;   // длина очередного фактора
    MinLen : integer;   // длина кратчайшего фактора
    Link : TLink;
begin
  Result:= 0;          // количество факторов
  MinLen:= MaxInt;     // длина кратчайшего фактора
  // количество вершин исходного графа:
  NodesCnt:= mGraph.mNodes.GetCount;
  // очистка полей mValue двудольного графа:
  Start:= NodeFirst as THamNode;
  while Assigned(Start) do begin
    Start.mValue:= 0;
    Start:= NodeNext as THamNode;
  end;
  // Пока не обработаны все вершины графа (исходного)
  while NodesCnt>0 do begin
    Inc(Result);  // Result= 1, 2, .. - цвет очередного контура
    CircLen:=0;   // длина текущего фактора (контура)
    // Ищем первую неокрашенную вершину в левой доле:
    Start:= NodeFirst as THamNode;
    while Assigned(Start) do begin
      with Start do if mLeft and (mValue = 0) then Break;
      Start:= NodeNext as THamNode;
    end;
    // Начиная с вершины Start обходим двудольный граф
    // отмечая цветом Result очередной замкнутый контур:
    // => вправо - по направлению текущего потока;
    // <= влево  - по обратным ссылкам mBack
    // пока не вернёмся к исходной вершине
    Left:= Start;
```

```
    repeat
      Dec(NodesCnt);         // счётчик оставшихся вершин
      Inc(CircLen);          // длина этого фактора (контура)
      Left.mValue:= Result; // метим контур текущим цветом
      // Left.mLink -- дуга из вершины Left в вершину Right
      // через которую течёт ненулевой поток (чёрный линк паросочетания)
      Link:= Left.mLink;
      // переход в левую долю вдоль потока:
      Left:= (Link.mDest as THamNode).mBack;
    until Left = Start;
    // После обхода контура запоминаем кратчайший:
    if CircLen < MinLen then begin
      MinLen:= CircLen;    // длина фактора (контура)
      aStart:= Start;      // стартовая вершина фактора (любая из)
    end;
  end; // while NodesCnt >0
end;


// Проверка возможности пропуска через двудольный граф
// насыщенного (максимального) потока
// Возвращает:
// FALSE -- если хотя бы одна из вершин не пропускает ни одной единицы потока
// TRUE  -- если через все вершины проходит хоть одна единица потока
// (Link.mHigh <> 0) -- признак закрытой дуги (заблокированной)

function THamGraph.Check: boolean;
var Node: THamNode;
    Link: TLink;
begin
  Result:= false;
  // Перебор вершин
  Node:= NodeFirst as THamNode;
  while Assigned(Node) do begin
    if Node.mLeft then begin
      // В левой доле исследуем исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.OutLinkNext;
      if not Assigned(Link) then Exit;
    end else begin
      // В правой доле исследуем входящие линки
      Link:= Node.InLinkFirst;
      while Assigned(Link) and (Link.mHigh<>0)
        do Link:= Node.InLinkNext;
      if not Assigned(Link) then Exit;
    end;
    Node:= NodeNext as THamNode;
  end;
  Result:= true;  // насыщенный поток возможен
end;

// Сохранение текущего потока в качестве наилучшего
// Вызывается при обнаружении очередного лучшего контура

procedure THamGraph.SaveBest(aBuf: TBuffer);
var Start, Node : THamNode;
begin
  aBuf.Clear; // очистка буфера результата
  // Начинаем с первой вершины в левой доле
  Start:= NodeFirst as THamNode;
  while not Start.mLeft do Start:= NodeNext as THamNode;
```

```
  // Обходим контур от Start до Start
  Node:= Start;
  repeat
    aBuf.Put(Node.mNode);
    Node:= Node.NextCircNode;
  until Node = Start;
end;

// Стягивание факторов в новый граф для вычисления прогноза стоимости
// aFact - количество факторов
// Вызывается после построения потока и факторов в последовательности:
// MarkMinPairsDicoty -> CalcFactors -> Shrink

function THamGraph.Shrink(aFact: integer): THugeGraph;
var i: integer;
    Link: TLink;
    N1, N2: TNodeInt;   // вершины стянутого графа помечены числами
    NH1, NH2: THamNode; // вершины данного двудольного графа
    Cost: integer;      // остаточная стоимость дуги
begin
  // Создаём пустой граф:
  Result:= THugeGraph.Create('',  // имя
                          true,   // граф направленный
                          false,  // вершины не нагружены
                          true    // дуги нагружены
                         );
  // Создаём и вставляем вершины, помеченные числами от 1 до aFact

  for i:= 1 to aFact do Result.InsertNode(TNodeInt.Create(i, 0, Result));

  // Соединяем дугами бесконечной длины все вершины созданного графа

  with Result do begin
    N1:= NodeFirst as TNodeInt;
    while Assigned(N1) do begin
      PosPush;
      N2:= NodeFirst as TNodeInt;
      while Assigned(N2) do begin
        if N1 <> N2 then SetLink(N1, N2, MaxInt);
        N2:= NodeNext as TNodeInt;
      end;
      PosPop;
      N1:= NodeNext as TNodeInt;
    end;
  end;

  // Находим минимальные относительные стоимости дуг нового графа
  // Остаточные стоимости дуг в полях TLink.mLow
  // определены при поиске минимального паросочетания (потока)
  // процедурой TGraph.MarkMinPairsDicoty

  NH1:= NodeFirst as THamNode;
  while Assigned(NH1) do begin
    // Обработка вершин левой доли:
    if NH1.mLeft then begin
      // Перебираем дуги вершины:
      Link:= NH1.OutLinkFirst;
      while Assigned(Link) do begin
        NH2:= (Link.mDest as THamNode).mBack;
        // Если линк ведёт в другой фактор (контур), то обрабатываем
        if NH1.mValue <> NH2.mValue then begin
          // Остаточная стоимость дуги после редуцирования
```

```
          // (определена процедурой TGraph.MarkMinPairsDicoty)
          Cost:= Link.mLow;
          // Находим линк между вершинами (факторами) в стянутом графе
          // Здесь поля NH1.mValue, NH2.mValue соответствуют факторам
          // и вершинам стянутого графа
          Link:= Result.GetLinkByNumber(NH1.mValue, NH2.mValue);
          // Назначаем этому линку пониженую (редуцированную) цену
          if Link.mValue > Cost then Link.mValue:= Cost;
        end;
        Link:= NH1.OutLinkNext;
      end;
    end;
    NH1:= NodeNext as THamNode;
  end;
  // Выполняем компрессию дуг, заменяя их длину кратчайшими расстояниями
  Result.Compress;
end;


// Оценка прогнозируемой стоимости и подсчёт количества факторов
// при текущем состоянии клапанов (открытых и закрытых дуг)
// Возвращает:
// Result   -- нижний предел стоимости для данного состояния клапанов mHigh
// aFactors -- количество факторов
// aStart   -- стартовую вершину кратчайшего фактора

function THamGraph.CalcCostAndFactors(
                           var aFactors: integer; //кол-во факторов
                           var aStart: THamNode   // стартовая вершина
                           ): integer;            // стоимость потока

var Level: integer; // уровень рекурсии

  // Рекурсивная процедура стягивания факторов

  procedure Local(aGraph: THamGraph);
  var Shrinked: TGraph; // очередной граф, образованный стягиванием факторов
      HamGr: THamGraph; // и соответствующий ему двудольный граф
      Cost: integer;    // стоимость потока
      Factors: integer; // количество факторов
      Dummy: integer;   // количество обнаруженных пар (не используем)
      Node: THamNode;   // любая вершина кратчайшего фактора
  begin

    Inc(Level); // отслеживаем уровень рекурсии

    if Level=1 then begin
      // На первом уровне проверим возможность достижения насыщенного потока
      if not aGraph.Check then begin
        // Насыщенный поток невозможен
        Result:= -1;   Exit;
      end;
    end;

    // Вычисляем стоимость полного потока на двудольном графе
    Cost:= aGraph.MarkMinPairsDicoty(Dummy);

    // Накапливаем результат (прогнозируемую стоимость)
    Inc(Result, Cost);

    // Находим факторы (контуры)
    Factors:= aGraph.CalcFactors(Node);
```

```
    // На первом уровне рекурсии запоминаем количество факторов
    // и любую вершину кратчайшего фактора

    if Level=1 then begin
      aFactors:= Factors;
      aStart:= Node;
    end;

    // Если фактор единственный, то выход с возвращением цены
    if Factors = 1 then Exit;

    // При наличии нескольких факторов (не связанных контуров)
    // стягиваем факторы в вершины, создаём новый полный граф
    // и компрессируем его дуги
    Shrinked:= aGraph.Shrink(Factors);

    // Создаём двудольный граф типа THamGraph
    // и рекурсивно повторяем предыдущие шаги

    HamGr:= THamGraph.Create(Shrinked);
    Local(HamGr);

    // Удаляем вспомогательные графы
    HamGr.Free;
    Shrinked.Free;
    Dec(Level); // уровень рекурсии -1
  end;

begin   { THamGraph.CalcCost }
  Result:=0;   aFactors:= 0; aStart:= nil;
  Level:= 0;
  Local(Self);
end;

// TClosed - элемент очереди, содержит:
// -- стоимость потока,
// -- уникальный идентификатор,
// -- множество закрытых дуг (клапанов),
// -- множество множеств отключаемых дуг кратчайшего фактора

type  TClosed = class (TItem)
        mCost: integer;  // стоимость потока
        mId: integer;    // уникальный идентификатор
        mClosed: TSet;   // множество закрытых дуг (клапанов)
        mFactors: TSet;  // множества отключаемых дуг кратчайшего фактора
        constructor Create(aCost: Integer; aClosed: TSet);
        destructor Destroy;  override;
        function Compare(arg: TItem): TCompare; override;
        procedure InsertLinksSet(aSet: TSet);
        procedure GatesOn;
        procedure GatesOff;
        procedure Print(var aFile: TextFile); override;
      end;

var ClosedId : integer; // уникальный идентификатор элемента очереди
                        // наращивается в конструкторе объекта

constructor TClosed.Create(aCost: Integer; aClosed: TSet);
begin
  inherited Create;
  Inc(ClosedId);    // наращиваем уникальный идентификатор
```

```
  mId:= ClosedId;    // и заносим в поле объекта
  mCost:= aCost;     // стоимость потока на момент создания объекта
  // Создаём и копируем множество закрытых дуг из параметра
  mClosed:= CreateSet;
  if Assigned(aClosed) then mClosed.CopyItems(aClosed);
  // Заготавливаем множество множеств закрываемых дуг
  // (по одному множеству для каждой вершины фактора).
  // В это множество будут вставляться множества закрываемых дуг
  // процедурой TClosed.InsertLinksSet
  mFactors:= CreateSet;
end;

destructor TClosed.Destroy;
begin
  mFactors.ClrAndDestroy;
  mFactors.Free;
  mClosed.Free;
  inherited;
end;

// Добавляет очередное множество отключаемых дуг aSet

procedure TClosed.InsertLinksSet(aSet: TSet);
begin
  mFactors.Insert(aSet)
end;

// Метод сравнения выстраивает объекты в приоритетной очереди
// в порядке не убывания стоимости,
// а при равной стоимости - в порядке вставки
// путём сравнения уникальных идентификаторов

function TClosed.Compare(arg: TItem): TCompare;
begin
  if mCost < (arg as TClosed).mCost
    then Result:= cmpLess
  else if mCost > (arg as TClosed).mCost
    then Result:= cmpGreate
  else { при равенстве стоимостей сравниваем уникальные идентификаторы }
    if mId < (arg as TClosed).mId
      then Result:= cmpLess
    else if mId > (arg as TClosed).mId
      then Result:= cmpGreate
    else Result:= cmpEq
end;

// Отключает дуги, содержащиеся в mClosed

procedure TClosed.GatesOff;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // дуга отключается единицей
      Link:= GetNext as TLink;
    end;
  end;
end;

// Включает дуги, содержащиеся в mClosed
```

```
procedure TClosed.GatesOn;
var Link: TLink;
begin
  with mClosed do begin
    Link:= GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0; // дуга подключается нулём
      Link:= GetNext as TLink;
    end;
  end;
end;

// Метод используется при отладке

procedure TClosed.Print(var aFile: TextFile);
var L : TLink;
begin
  Writeln(aFile);  Writeln(aFile, '=== C L O S E D === ', mCost);
  L:= mClosed.GetFirst as TLink;;
  while Assigned(L) do begin
    Write(aFile, (L.mOwner as THamNode).mNode.GetName);
    Write(aFile, ' -> ');
    Writeln(aFile, (L.mDest as THamNode).mNode.GetName);
    L:= mClosed.GetNext as TLink;;
  end;
end;

//            Метод TGraph.GenHamilton
// Ускоренный приближённый метод отыскания Гамильтоновых контуров

function TGraph.GenHamilton(var aCost: integer): TBuffer;

var   HamGraph: THamGraph; // вспомогательный двудольный граф
(Dicotyledonous)
      Que : TSet;          // Приоритетная очередь клапанов (по неубыванию)
      BestCost: integer;   // лучшая цена на текущий момент
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Отключение дуг, переданных в множестве aLinks

  procedure LinksOff(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 1; // отключить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Включение дуг, переданных в множестве aLinks

  procedure LinksOn(aLinks: TSet);
  var Link: TLink;
  begin
    Link:= aLinks.GetFirst as TLink;
    while Assigned(Link) do begin
      Link.mHigh:= 0;  // включить дугу
      Link:= aLinks.GetNext as TLink;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
// Обработка объекта с клапанами aClosed
// (множеством отключенных и множествами отключаемых дуг)

procedure Handle(aClosed: TClosed);
var Cost: integer;    // прогнозируемая или точная стоимость решения
    Factors: integer; // количество факторов в очередном решении
    Start: THamNode;  // стартовая вершина кратчайшего фактора
    Closed: TClosed;  // новый объект с клапанами
    Links: TSet;      // множество откюченных дуг в текущий момент
    Node: THamNode;
begin
  // Отключить блокированные дуги объекта
  aClosed.GatesOff;

  // Цикл: поочерёдно отключатем дуги, ведущие внутрь фактора,
  // оцениваем стоимость соответствующих потоков,
  // и ставим в приоритетную очередь элементы TClosed
  // для частичных решений

  // Перебор множеств отключаемых дуг:
  Links:= aClosed.mFactors.GetFirst as TSet;
  while Assigned(Links) do begin
    // Отключаем дуги одной из вершин фактора, ведущие внутрь этого фактора
    LinksOff(Links);
    // При текущем состоянии клапанов определить стоимость потока
    Cost:= HamGraph.CalcCostAndFactors(Factors, Start);
    // Вновь подключаем дуги вершины, ведущие внутрь фактора
    LinksOn(Links);
    // Если поток существует, и он лучше текущего...
    if (Cost>=0) and (Cost< BestCost) then begin
      // Здесь поток существует и его стоимость ниже текущего минимума.
      // Если количество факторов=1, то запомнить лучший результат.
      if Factors=1 then begin
        BestCost:= Cost;            // лучшая цена
        aCost:= Cost;               // она же как результат
        // сохраняем последовательность вершин контура в буфере результата
        HamGraph.SaveBest(Result);
      end else begin
        // Здесь количество факторов Factors > 1 (частичное решение),
        // то создаём и ставим в очередь множество закрытых клапанов
        Links.Add(aClosed.mClosed); // объединяем Links и mClosed
        Closed:= TClosed.Create(Cost, Links);
        // Обходим кратчайший фактор
        // и сохраняем в объекте Closed отключаемые дуги
        Node:= Start; // Начальная (любая) вершина фактора
        repeat
          // Функция THamNode.GenFactorsLinks создаёт множество дуг
          // ведущих внутрь текущего фактора
          Closed.InsertLinksSet(Node.GenFactorsLinks);
          Node:= Node.NextCircNode; // следующая вершина фактора
        until Node = Start;
        Que.Insert(Closed); // Вставляем объект в приоритетную очередь
      end;
    end;
    // Следующее множество отключаемых дуг:
    Links:= aClosed.mFactors.GetNext as TSet
  end;
  // Вернуть исходное состояние клапанов
  aClosed.GatesOn;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
var  Closed: TClosed; // очередное множество закрытых клапанов

begin  { TGraph.GenHamilton }

  Result:= TBuffer.Create;  // создаём пустой буфер для хранения решения
  aCost:= -1;               // стоимость на случай отсутствия решения

  // Создаём вспомогательный двудольный граф HamGraph (Dicotyledonous)
  // с целью определения минимальной стоимости полного потока,
  // а также очередь клапанов Que (элементов типа TGate)

  HamGraph:= THamGraph.Create(Self);
  Que:= CreateSet;  // Приоритетная очередь клапанов по неубыванию стоимости
  BestCost:= MaxInt;// Минимальная стоимость на текущий момент

  // Создаём и вставляем в приоритетную очередь фиктивный элемент,
  // в котором множества закрытых и закрываемых линков пусты
  Closed:= TClosed.Create(0, nil);
  Closed.InsertLinksSet(CreateSet);
  Que.Insert(Closed);

  // Цикл обработки приоритетной очереди:
  while Que.GetCount > 0 do begin
    // Выбрать из очереди объект-множество клапанов
    Closed:= Que.GetFirst as TClosed;
    // Если его прогнозируемая цена меньше лучшей на данный момент,
    // то обработать это состояние клапанов
    if Closed.mCost < BestCost then Handle(Closed);
    // Удалить объект-множество клапанов
    Que.Delete(Closed);
    Closed.Free;
  end;

  // Очистка памяти:
  Que.Free;       // приоритетная очередь клапанов по неубыванию стоимости
  HamGraph.Free;  // вспомогательный двудольный граф (Dicotyledonous)
end;


// Быстрый приближённый алгоритм поиска гамильтонова контура
// на основе поиска кратчайшей гамильтоновой цепи
// в неориентированном графе

function TGraph.GenHamilton_Quick(var aCost: integer): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Обработка очередной вершины:
  // возвращаются две ближайшие вершины (aY, aZ)
  // и сумма расстояний

  function FindX(aX : TNode; var aY, aZ : TNode): integer;
  var L: TLink;
      YVal, ZVal : integer; // расстояния к соседям
  begin
    aY:= nil; aZ:= nil;
    YVal:= MaxInt; ZVal:= MaxInt;
    // Перебор исходящих линков:
    L:= aX.OutLinkFirst;
    // Всегда YVal <= ZVal
    while Assigned(L) do begin
      if L.mValue <= YVal then begin
        ZVal:= YVal;
```

```
        YVal:= L.mValue;
        aZ:= aY; aY:= L.mDest;
      end else if L.mValue < ZVal then begin
        ZVal:= L.mValue;
        aZ:= L.mDest;
      end;
      L:= aX.OutLinkNext;
    end;
    // Возвращаем сумму расстояний к ближайшим соседям
    Result:= YVal + ZVal;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Поиск вершины с минимальной суммой расстояний
  // к двум ближайшим соседям

  function FindNode(var aX, aY, aZ : TNode): integer;
  var X, Y, Z : TNode;
      Cost, BestCost: integer;
  begin
    aX:= nil; aY:= nil; aZ:= nil;
    BestCost:= MaxInt;
    X:= NodeFirst;
    while Assigned(X) do begin
      Cost:= FindX(X, Y, Z);
      if Cost < BestCost then begin
        BestCost:= Cost;
        aX:= X; aY:= Y; aZ:= Z;
      end;
      X:= NodeNext;
    end;
    Result:= BestCost;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

var NodeX, NodeY, NodeZ : TNode;
    Cost, CostX : integer;

begin  { GenHamilton_Quick }

  // Для орграфа используем основной метод:
  if mDirect then begin
    Result:= GenHamilton(aCost);
    Exit;
  end;

  // Находим вершину X и двух её ближайших соседей Y, Z таких,
  // что сумма расстояний от X к соседям Y и Z минимальна

  CostX:= FindNode(NodeX, NodeY, NodeZ);

  // Строим кратчайшую цепь от вершины Y к вершине Z
  // с исключением вершины X

  Result:= GenHamPathStrip(NodeY, NodeZ, NodeX, Cost);
  if Cost < 0 then Exit;

  aCost:= Cost + CostX;  // общая стоимость контура

  // Вставляем вершину в разрез между Z и Y, замыкая тем самым контур
  Result.Push(NodeX);

  // Вращаем буфер, пока первая вершина графа не сместится в его начало
```

```
  NodeX:= NodeFirst;
  repeat
    NodeY:= Result.Get as TNode;
    if NodeY <> NodeX then Result.Put(NodeY);
  until NodeY = NodeX;
  Result.Push(NodeY);
end;


//#########################################################################
//                                                                        #
//                          Цепь Гамильтона                               #
// Кристофидес, стр 273,                                                  #
// метод штрафования верши модифицированый                                #
//                                                                        #
//#########################################################################

// Вспомогательная функция для вычисления стоимости линка
// с учётом штрафов инцидентных вершин

function CalcValue(aLink: TLink): integer;
begin
  with aLink do Result:= mValue + mOwner.mDist + mDest.mDist;
end;


type  // THamLink -- вспомогательный линк для построения остова

  THamLink = class (TItem)
    mValue: integer; // стоимость с учётом штрафов
    mLink: TLink;    // ссылка на исходный линк
    constructor Create(aLink: TLink);
    function  Compare(arg: TItem): TCompare; override;
    procedure Print(var aFile: TextFile); override;
  end;


constructor THamLink.Create(aLink: TLink);
begin
  inherited Create;
  mLink:= aLink;
  // Условная стоимось формируется с учётом штрафов
  mValue:= CalcValue(aLink);
end;


function THamLink.Compare(arg: TItem): TCompare;
begin
  Result:= inherited Compare(arg);
  if Result = cmpEq then Exit;
  with mLink do begin
    // Для неориентированного графа сравниваем исходную и конечную вершины
    // на предмет встречных линков
    if not mOwner.mOwner.mDirect and  // если не орграф
      (mOwner=(arg as THamLink).mLink.mDest) and
      (mDest=(arg as THamLink).mLink.mOwner)
      then begin
        // Здесь линки направлены встречно, отвергаем дубликат:
        Result:= cmpEq;    Exit;
      end;
  end;
```

```
  // Если линки не совпадают, то сортируем по неубыванию стоимости
  Result:= cmpLess;
  if mValue > (arg as THamLink).mValue then Result:= cmpGreate;
end;

// В орграфе выводим по направлению стрелок,
// а в графе - по алфавиту

procedure THamLink.Print(var aFile: TextFile);
var N1, N2 : TNode;
begin
  with mLink do begin
    N1:=mOwner;  N2:=mDest;
    if not mOwner.mOwner.mDirect and
       (mOwner.Compare(mDest) = cmpGreate) then begin
      N1:=mDest;   N2:=mOwner;
    end;
    Write(aFile, ' ' + N1.GetName + N2.GetName);
    Write(aFile, '=', mValue:3, '':1);
  end;
end;


// Поиск гамильтоновой цепи из вершины aStart в вершину aFin

function TGraph.GenHamPath(aStart,        // начальная вершины
                           aFin: TNode;  // конечная вершины
                           var aCost: integer  // длина цепи
                           ): TBuffer;
begin
  if mDirect
    // Для орграфа используется основной метод поиска гамильтонова контура
    then Result:= GenHamPathDir(aStart, aFin, aCost)
    // Для неориентированного графа - метод штрафования верши
    // Кристофидес, стр 273,
    // метод штрафования верши модифицированый
    else Result:= GenHamPathStrip(aStart, aFin, nil, aCost);
end;


// Поиск гамильтоновой цепи из вершины aStart в вершину aFin
// в ориентированном графе

function TGraph.GenHamPathDir(aStart, aFin: TNode; var aCost: integer):
TBuffer;
var Closer,      // Замыкающая вершина
    Node: TNode;
begin
  Closer:= TNode.Create(0, Self);  // создаём вспомогат. замыкающую вершину
  InsertNode(Closer);              // вставляем её в граф
  Closer.MakeLink(aStart, 0);      // создаём связь Closer -> aStart
  aFin.MakeLink(Closer, 0);        // создаём связь aFin -> Closer
  Result:= GenHamilton(aCost);     // находим контур Гамильтона
  // Удаляем из буфера результата замыкающую вершину
  repeat
    Node:= Result.Get as TNode;
    if Node = Closer then Break;
    Result.Put(Node);
  until false;
  aFin.RemoveLink(Closer);     // удаляем связь aFin -> Closer
  Closer.RemoveLink(aStart);   // удаляем связь Closer -> aStart
  RemoveNode(Closer);          // удаляем замыкающую вершину из графа
```

```
  Closer.Free;                  // уничтожаем вершину
end;


// Поиск гамильтоновой цепи из вершины aStart в вершину aFin
// с исключением вершины aStrip
// (если aStrip=nil, то без исключения)
// Кристофидес, стр 273,
// метод штрафования верши модифицированый

function TGraph.GenHamPathStrip(aStart,        // начальная вершина
                                aFin,          // конечная вершина
                                aStrip: TNode; // исключаемая вершина
                                var aCost: integer  // длина цепи
                                ): TBuffer;

// Поля вершин TNode используются так:
// mDist    -- текущий штраф
// mPower   -- текущая степень вершины в остовном дереве
// mColor   -- текущая окраска
// mRoot    -- принадлежность поддереву
// mFlow    -- для временного хранения приращения штрафа
// mLimit   -- обратный счётчик штрафования (до нуля)

var   Que : TSet; // очередь рёбер по неубыванию стоимости с учётом штрафов
     Tree : TSet; // множество чёрных рёбер покрывающего дерева
   Marked : TBuffer;  // очередь вершин для маркировки поддеревьев
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка штрафных полей mDist (вызывается единожды)

  procedure ClearFines;
  var Node: TNode;      // текущая вершина
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      with Node do begin
        mDist:= 0;    // штраф
        mLimit:= 1;   // ненулевой предел на штрафование вершины
        mRoot:= nil;  // корневая метка
      end;
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Возобновление счётчиков штрафования Node.mLimit
  // Вызывается после уменьшения невязки или стоимости

  procedure SetLimits(aLimit: integer);
  var Node: TNode;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mLimit:= aLimit; // предел на штрафование вершины (счётчик)
      Node:= NodeNext;
    end;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Очистка цвета вершин и рёбер, а также степеней вершин
  // (вызывается на каждой итерации перед построением дерева)
  // mColor = CWhite   -- цвет белый
  // mPower = 0        -- степень вершины в дереве
```

```
  procedure ResetNodesAndLinks;
  var Node : TNode;
      Link : TLink;
  begin
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mColor:= CWhite; // цвет
      Node.mPower:= 0;      // степень вершины в покрывающем дереве
      // Сбросить цвета рёбер
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        Link.mColor:= CWhite;
        Link:= Node.OutLinkNext;
      end;
      Node:= NodeNext;
    end;
    // Исключаемую вершину пометить красным:
    if Assigned(aStrip) then aStrip.mColor:= CRed;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Локальная процедура построения минимального покрывающего дерева
  // Формирует множество чёрных рёбер дерева и помещает в множество Tree

  function GenCover: boolean;

  var Node : TNode;     // текущая вершина
      Link : TLink;     // для пометки встречного линка
        HL : THamLink;  // ребро для дерева
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Добавление в очередь Que линков к соседним белым вершинам.
    // Линки HL сортируются в Que по неубыванию штрафной длины

    procedure AddTreeLinks(aNode: TNode);
    var  Link : TLink;
           HL : THamLink;
    begin
      // Обработка исходящих связей
      Link:= aNode.OutLinkFirst;
      while Assigned(Link) do begin
        // Вставить только линки, ведущие к белым вершинам
        if Link.mDest.mColor = CWhite then begin
          // Сконструировать вспомогательный линк:
          HL:= THamLink.Create(Link);
          // И вставить в очередь:
          if not Que.Insert(HL) then HL.Free;
        end;
        Link:= aNode.OutLinkNext;
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  var N : integer;  // количество присоединяемых вершин

  begin  { GenCover }
    Result:= true;
    // Предварительная очистка:
    Tree.ClrAndDestroy;   // дерево результата
    ResetNodesAndLinks;   // очистка вершин и рёбер: mColor = CWhite
    Node:= aStart;        // начинаем с исходной вершины
    Node.mColor:= CBlack; // красим чёрным
    // Присоединить все вершины графа кроме красной
    // (линков будет на единицу меньше)
    N:= mNodes.GetCount-1;
```

```
    if Assigned(aStrip) then Dec(N); // красную обойти стороной
    while Assigned(Node) and (Tree.GetCount < N) do  begin
      // Добавить в очередь Que линки к ближайшим белым вершинам:
      AddTreeLinks(Node);
      // Перебрать линки Que в поиске ближайшей белой вершины
      Node:= nil;                     // искомая вершина пока не найдена
      HL:= Que.GetFirst as THamLink;  // первый линк -- кратчайший
      while Assigned(HL) do begin
        Que.Delete(HL);               // удалить линк из очереди
        // Если линк ведёт к белой вершине, то ближайшая найдена
        with HL.mLink do if mDest.mColor = CWhite
            then Node:= mDest;   // приёмник связи
        if Assigned(Node) then begin
          // Здесь ближайшая белая вершина найдена:
          Node.mColor:= CBlack;     // присоединить её к множеству чёрных
          Tree.Insert(HL);          // вставить линк в результат (дерево)
          HL.mLink.mColor:= CBlack; // и отметить чёрным
          // Отметить чёрным и встречный линк:
          Link:= HL.mLink.GetReverse;
          if Assigned(Link) then Link.mColor:= CBlack;
          Break; // выход из while
        end else begin
          // Этот линк не ведёт к белой вершине, уничтожить его
          HL.Free;
          HL:= Que.GetNext as THamLink; // и взять следующий
        end;
      end; // while
    end; // while
    // Если остов не построен, то граф не связан, и решения не существует
    if Tree.GetCount < N then begin
      Tree.ClrAndDestroy;
      Result:= false;
    end; // if
    Que.ClrAndDestroy;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Подсчёт степеней всех вершин, помещение их в поля TNode.mPower
  // и подсчёт невязки через количество листьев

  function CalcPowerNodes: integer;
  var HL : THamLink;   // текущий линк дерева Tree
      Node: TNode;
  begin
    // Предварительно очистить поля mPower
    Node:= NodeFirst;
    while Assigned(Node) do begin
      Node.mPower:= 0;  // степень вершины в покрывающем дереве
      Node:= NodeNext;
    end;
    // Перебор линков в дереве Tree:
    HL:= Tree.GetFirst as THamLink;
    while Assigned(HL) do begin
      // наращиваем степени вершин дерева :
      Inc(HL.mLink.mOwner.mPower);     // в источнике дуги
      Inc(HL.mLink.mDest.mPower);      // в приёмнике дуги
      HL:= Tree.GetNext as THamLink;
    end;
    // Подсчитать невязку как сумму листьев минус 2
    Result:=-2;
    Node:= NodeFirst;
    while Assigned(Node) do begin
      if Node.mPower=1 then Inc(Result);
```

```
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Маркировка поддерева из данной вершины (метим поле mRoot).
// Дерево обходим в ширину по чёрным линкам.
// Маркированные аершины красим серым.

procedure Mark(aNode: TNode);
var   Link : TLink;
      Node : TNode;
begin
  Marked.Clear;          // очередь вершин для маркировки поддеревьев
  Marked.Put(aNode);     // занести в очередь корневую вершину
  aNode.mColor:= CGray; // и окрасить серым
  // Пока очередь не пуста:
  while Marked.GetCount > 0 do begin
    Node:= Marked.Get as TNode;    // выбрать вершину из очереди
    Node.mRoot:= aNode;            // и отметить её в поле mRoot
    Link:= Node.OutLinkFirst;
    // Поместить в очередь соседние вершины, связанные чёрными линками:
    while Assigned(Link) do begin
      // Если линк чёрный
      if (Link.mColor = CBlack) and
         // и цвета инцидентных вершин не совпадают
         (Node.mColor <> Link.mDest.mColor) then begin
        // то присоединить соседа:
        Marked.Put(Link.mDest);      // в очередь
        Link.mDest.mColor:= CGray;  // и окрасить серым
      end;
      Link:= Node.OutLinkNext;
    end;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисление минимального штрафа в результате разрыва ребра aLink
// Вызывается из CalcNodeFine
// Возвращает штраф и признак успеха aStatus

function CalcLinkFine(aLink: TLink; var aStatus: boolean): integer;
var Link : TLink;
    Node : TNode;
    Fine : integer;
begin
  Result:= MaxInt;  aStatus:= false;
  // В текущий момент все вершины окрашены чёрным, а линки так:
  // принадлежащие дереву - чёрным, разорваный - серым, прочие - белым
  // Пометить вершины двух поддеревьев (поля mRoot)
  Mark(aLink.mOwner);  // поддерево mOwner
  Mark(aLink.mDest);   // поддерево mDest
  // Найти ребро минимального веса, соединяющее два поддерева
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Восстановить цвет после маркировки (кроме красной вершины):
    if Node <> aStrip then Node.mColor:= CBlack;
    // Если это вершина поддерева mOwner и она не корневая
    if (Node.mRoot = aLink.mOwner) and (Node <> aLink.mOwner) then begin
      // то обрабатать её, перебирая исходящие линки
      Link:= Node.OutLinkFirst;
      while Assigned(Link) do begin
        // Если ребро ведёт в другое поддерево,
        // то запомнить его минимальный вес с учётом штрафа:
```

```
        // Link.mDest.mRoot -- метка вершины
        // aLink.mDest -- корень оторванной части
        if Link.mDest.mRoot = aLink.mDest then begin
          Fine:= CalcValue(Link);
          if Result > Fine then begin
            Result:= Fine;  // предварительный результат
            aStatus:= true; // признак наличия результата
          end;
        end;
        Link:= Node.OutLinkNext;
      end;
    end;
    Node:= NodeNext;
  end;
  // Окончательный результат (приращение штрафа)
  // вычислить как разность
  if aStatus then Result:= Result - CalcValue(aLink);
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Подсчёт минимального приращения штрафа для данной вершины
// вызывается из CalcFines
// Возвращает штраф и признак успеха aStatus

function CalcNodeFine(aNode: TNode; var aStatus: boolean): integer;
var Link, Reverse: TLink;  // прямой и обратный линки
    Fine: integer;         // вычисленный штраф
    OK: boolean;           // признак успеха
begin
  Result:= MaxInt;  aStatus:= false;
  // Поочерёдно "разрываем" линки вершины, входящие в состав дерева
  // (чёрные линки) и строим обходные пути,
  // соединяющие разорванные части дерева
  Link:= aNode.OutLinkFirst;
  while Assigned(Link) and (Result > 0) do begin
    if Link.mColor = CBlack then begin
      // Этот линк принадлежит дереву
      // Временно "разорвать" его и встречный линк (красить серым)
      Link.mColor:= CGray;
      Reverse:= Link.GetReverse;
      if Assigned(Reverse) then Reverse.mColor:= CGray;
      // Вычислить штраф:
      aNode.OutPosPush;   // сохранить позицию перебора
      Fine:= CalcLinkFine(Link, OK); // вычислить штраф
      aNode.OutPosPop;    // восстановить позицию перебора
      // Вновь восстанавить цвета прямого и встречного линков
      Link.mColor:= CBlack;
      if Assigned(Reverse) then Reverse.mColor:= CBlack;
      // Запоминаем минимальное приращение штрафа
      if OK and (Result > Fine) then begin
        Result:= Fine;
        aStatus:= true;
      end;
    end;
    Link:= aNode.OutLinkNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Подсчёт и установка штрафов для всех вершин
// Возвращает количество оштрафованных вершин

function CalcFines: integer;
var Node: TNode;
```

```
    Fine: integer;
    OK: boolean;
begin
  Result:=0; // для подсчёта оштрафованных вершин
  // Сканировать вершины, степень которых превышает 2,
  // вычислять приращения штрафов и временно помещать в поля mFlow
  // Node.mPower -- степень вершины в дереве
  // Node.mFlow -- сюда временно помещаем приращение штрафа
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Node.mFlow:= 0;          // mFlow -- приращение штрафа
    if (Node.mPower > 2)     // если это узел
       and (Node.mLimit > 0)// и лимит штрафования не исчерпан
      then begin
      Dec(Node.mLimit); // уменьшить лимит
      Inc(Result);      // нарастить счётчик оштрафованных
      // Вычисляить минимальный штраф, уменьшающий её степень:
      PosPush; // сохранить позицию перебора вершин
      Fine:= CalcNodeFine(Node, OK);
      PosPop;  // восстановить позицию перебора вершин
      if OK then begin
        if Fine = 0 then Fine:=1;  // реальный штраф больше нуля
        Node.mFlow:= Fine;         // mFlow -- приращение штрафа
      end;
    end; // if
    Node:= NodeNext;
  end; // while
  // Нарастить штрафы вершин mDist:= mDist + Node.mFlow
  Node:= NodeFirst;
  while Assigned(Node) do begin
    Inc(Node.mDist, Node.mFlow);
    Node:= NodeNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вычисление минимального приращения цены
// в результате разрыва ребра aLink
// Вызывается из CalcDeltaNode
// Возвращает кратчайший белый линк, соединяющий листья

function CalcDeltaLink(aLink: TLink; var aRes: TLink): integer;
var Link : TLink;
    Node : TNode;
begin
  Result:= MaxInt;
  aRes:= nil;
  // В текущий момент все вершины окрашены чёрным,
  // линки дерева - чёрным, разорваный линк - серым, остальные - белым
  // Пометить вершины двух поддеревьев (поля mRoot)
  Mark(aLink.mOwner);  // поддерево mOwner
  Mark(aLink.mDest);   // поддерево mDest
  // Найти линк минимального веса, соединяющий два поддерева
  // через вершины со степенями 1 (листья)
  Node:= NodeFirst;
  while Assigned(Node) do begin
    // Восстановить цвет после маркировки (кроме красной вершины)
    if Node <> aStrip then Node.mColor:= CBlack;
    // Если это лист в дереве mOwner
    if (Node.mRoot = aLink.mOwner) and (Node.mPower < 2)
      then begin
      // то обработать его
      Link:= Node.OutLinkFirst;
```

```
        while Assigned(Link) do begin
          // Запомнить кратчайший линк к листу другого поддерева
          if (Link.mDest.mRoot = aLink.mDest)  // в другом поддереве
            and (Link.mDest.mPower < 2)        // и степень меньше 2-х
            and (Result > CalcValue(Link))     // и штрафная длина меньше
          then begin
              Result:= CalcValue(Link);  // промежуточный результат
              aRes:= Link;               // белый линк
          end; // if
          Link:= Node.OutLinkNext;
        end; // while
      end; // if
      Node:= NodeNext;
    end; // while
    // Окончательный результат вычислить как разность
    if Assigned(aRes) then Result:= Result - CalcValue(aLink);
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Поиск подходящей пары линков для снижения степени узла aNode
// Вызывается из ForceCoverTree
// Возвращает два линка: aBlack -- разрываемый, aWhite -- замещающий

procedure FindBlackAndWhite(aNode: TNode; var aBlack, aWhite: TLink);
var Link : TLink;   // прямой линк
    White: TLink;   // белый линк
    Delta: integer; // текущее приращение
    Minim: integer; // минимальное приращение
begin
  aBlack:= nil;    aWhite:= nil;
  Minim:= MaxInt;
  // Поочерёдно "разрывать" линки вершины, входящие в состав дерева
  // (эти линки помечены чёрным)
  // и отыскивать обходные пути,
  // соединяющие разорванные части дерева через листья
  Link:= aNode.OutLinkFirst;
  while Assigned(Link) do begin
    if Link.mColor = CBlack then begin
      // Этот линк принадлежит дереву, разорвать его (красим серым)
      Link.mColor:= CGray;
      // При разрыве уменьшаются степени инцидентных вершин:
      with Link do begin
        Dec(mOwner.mPower);
        Dec(mDest.mPower);
      end;
      aNode.OutPosPush; // сохр. позицию перебора
      // Вычислить приращение цены
      Delta:= CalcDeltaLink(Link, White);
      aNode.OutPosPop;  // восст. позицию перебора
      // Восстанавить цвет линка
      Link.mColor:= CBlack;
      // При воссоединении чёрного линка
      // восст. степени инцидентных вершин:
      with Link do begin
        Inc(mOwner.mPower);
        Inc(mDest.mPower);
      end;
      // Запомнить минимальное приращение
      if Assigned(White) and (Delta < Minim) then begin
        Minim:= Delta;   // приращение
        aBlack:= Link;   // чёрный линк (в дереве)
        aWhite:= White;  // белый линк (вне дерева)
      end;
```

```
    end;
    Link:= aNode.OutLinkNext;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Удаление из дерева вспомогательных линков

procedure Tree_Delete(aLink: TLink);
var HL : THamLink; // линк в дереве Tree
    Rev: TLink;    // встречный линк
begin
  // Уменьшить степени инцидентных вершин чёрного ребра aLink:
  Dec(aLink.mOwner.mPower);
  Dec(aLink.mDest.mPower);
  // Прямой и встречный линк метить белым
  aLink.mColor:= CWhite;
  Rev:= aLink.GetReverse;  // встречный линк
  if Assigned(Rev) then Rev.mColor:= CWhite;
  // Найти в дереве соответствующий вспомогательный линк
  HL:= Tree.GetFirst as THamLink;
  while Assigned(HL) do begin
    if (HL.mLink = aLink) or (HL.mLink = Rev) then Break;
    HL:= Tree.GetNext as THamLink;
  end;
  // Удалить его из дерева и ликвидировать
  if Assigned(HL) then begin
    Tree.Delete(HL);
    HL.Free;
  end;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Вставка в дерево вспомогательных линков

procedure Tree_Insert(aLink: TLink);
var HL : THamLink;
    Rev: TLink;
begin
  // Увеличить степени инцидентных вершин:
  Inc(aLink.mOwner.mPower);
  Inc(aLink.mDest.mPower);
  // Прямой и встречный линк метить чёрным
  aLink.mColor:= CBlack;
  Rev:= aLink.GetReverse;
  if Assigned(Rev) then Rev.mColor:= CBlack;
  // Сконструировать вспомогательный линк:
  HL:= THamLink.Create(aLink);
  // И вставить его в дерево:
  if not Tree.Insert(HL) then HL.Free;
end;
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Принудительное уменьшение степеней узлов остова

procedure ForceCoverTree;
var Node: TNode;
    Black, White : TLink;
begin
  // Обработать узлы (вершины со степенями более 2)
  // Node.mPower -- степень вершины в дереве
  Node:= NodeFirst;
  while Assigned(Node) do begin
    if Node.mPower > 2 then begin
      // Вычислить минимальный штраф, уменьшающий её степень
```

```
        PosPush; // сохранить позицию перебора вершин
        FindBlackAndWhite(Node, Black, White);
        PosPop;  // восстановить позицию перебора вершин
        // Если пара рёбер обнаружена
        if Assigned(Black) and Assigned(White) then begin
          // Видоизменить дерево
          Tree_Delete(Black);   // удалить из дерева разрываемый линк
          Tree_Insert(White);   // вставить в дерево соединяющий линк
        end;
      end;
      Node:= NodeNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Форсированное преобразование дерева в цепочку
  // с вычислением стоимости цепочки

  function Force(var aCost: integer): TBuffer;

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Локальная функция для переноса цепи
    // из буфера aBuf в результат Result
    // с одновременным подсчётом стоимости цепи

    function MoveResult(aRes: TBuffer): integer;
    var HL : THamLink; // текущий линк дерева из Tree
    begin
      Result:= 0;
      aRes.Clear;
      // Перебор линков в покрывающем дереве:
      HL:= Tree.GetFirst as THamLink;
      while Assigned(HL) do begin
        aRes.Put(HL.mLink);
        Inc(Result, HL.mLink.mValue); // накопление стоимости
        HL:= Tree.GetNext as THamLink;
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var  Delta : integer;  // невязка, вычисляемая по числу листьев

  begin { Force }
    // Повторять, пока невязка больше нуля
    repeat
      // Форсировано снизить степени "узлов"
      ForceCoverTree;
      // Подсчитать степени вершин и невязку через количество листьев
      Delta:= CalcPowerNodes;
    until Delta=0;
    // Перенести цепь из буфера Buf в результат Result
    // и подсчитать стоимость цепи aCost
    Result:= TBuffer.Create;
    aCost:= MoveResult(Result);  //
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  // Преобразование буфера результата:
  // линки заменяются цепочкой вершин от Start до Fin

  function ConvertLinksToNodes(aLinks: TBuffer): TBuffer;

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Поиск в буфере aLinks следующей за aNode вершины

    function FindNext(aNode: TNode): TNode;
    var Link: TLink;
        i: integer;
    begin
      Result:= nil;
      // Круговое вращение буфера в поиске нужного линка
      for i:= 1 to aLinks.GetCount do begin
        Link:= aLinks.Get as TLink;
        // Если линк найден, то цикл прекращается
        with Link do begin
          if aNode = mOwner then begin
            Result:= mDest;  Break;
          end;
          if aNode = mDest then begin
            Result:= mOwner; Break;
          end;
        end;
        // Если не тот линк, то вернуть его в буфер
        aLinks.Put(Link);
      end;
    end;
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  var  Node: TNode;

  begin { ConvertLinksToNodes }

    Result:= TBuffer.Create;
    // Цепочка начинается с вершины aStart
    Node:= aStart;
    repeat
      Result.Put(Node);       // поместить в буфер
      Node:= FindNext(Node);  // и найти следующую
    until (Node = aFin) or not Assigned(Node);
    // Цепочка завершается вершиной aFin
    if Assigned(Node) then Result.Put(Node);
    aLinks.ClrAndDestroy;
    aLinks.Free;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    OK : boolean;        // признак связности графа
    Changed: boolean;    // признак снижения стоимости или невязки
    Delta : integer;     // невязка, вычисляемая по степеням вершин
    MinDelta : integer; // текущая минимальная невязка
    Nodes: integer;      // количество оштрафованных вершин
    Cost: integer;       // текущая стоимость
    BestCost: integer;   // стоимость лучшего промежуточного решения
    Res: TBuffer;        // очередное решение (цепь)
    FineLimit: integer; // предел штрафования вершины (счётчик)

begin  { TGraph.GenHamPath }

  aCost:= -1; // цена на случай отсутствия решения
```

```
// Создать буфер для линков гамильтоновой цепи:
Result:= TBuffer.Create;
// Если это орграф, то выход:
if mDirect then Exit;

// Назначить начальное значение для счётчиков штрафования вершин
FineLimit:= mNodes.GetCount;

// Создать:
Que:= CreateSet;          // множество для очереди линков
Tree:= CreateSet;         // буфер для линков дерева
Marked:= TBuffer.Create;  // очередь вершин для маркировки

BestCost:= MaxInt;  // начальная лучшая стоимость

// Очистить штрафные поля mFines:
ClearFines;

// Назначить максимальные штрафы (mDist) для крайних вершин:
aStart.mDist:= MaxInt div 4;
aFin.mDist:= aStart.mDist;

MinDelta:= MaxInt;       // минимальная невязка
Nodes:= 0;               // количество оштрафованных вершин

// Повторять, пока существуют оштрафованные вершины
repeat
  // Построить кратчайшее покрывающее дерево с учётом штрафов:
  OK:= GenCover;
  // Выход, если граф не связан:
  if not OK then Break;
  // Подсчитать невязку Delta как сумму листьев - 2
  Delta:= CalcPowerNodes;
  // Подсчитать новые штрафов
  // на основе текущих степеней вершин (полей mPower)
  // вернуть количество оштрафованных вершин Nodes
  Nodes:= CalcFines;
  Changed:= false;   // признак снижения стоимости или невязки
  // Искать форсированное решение
  Res:= Force(Cost);
  // Если оно лучше текущего, то запомнить, а иначе удалить
  if Cost < BestCost then begin
    Changed:= true;   // признак снижения стоимости или невязки
    BestCost:= Cost;  // запомнить лучшую стоимость
    aCost:= Cost;     // и текущую стоимость
    Result.Free;      // прежний результат удалить
    Result:= Res;     // а новый сохранить
  end else Res.Free;  // а иначе удалить новый результат
  // При уменьшении невязки
  if Delta < MinDelta then begin
    Changed:= true;       // признак снижения стоимости или невязки
    MinDelta:= Delta;     // запомнить минимальную невязку
  end;
  // Если уменьшилась стоимость или невязка
  if Changed then begin
    SetLimits(FineLimit); // возобновить количество попыток штрафования
    FineLimit:= 2 + 3*FineLimit div 4; // пересчитать лимит штрафования
  end;
until Nodes = 0;  // прекратить, если штрафование не выполнялось
// По окончании преобразовать буфер результата:
// линки --> в вершины от Start ... до Fin
```

```
  if OK then Result:= ConvertLinksToNodes(Result);
  // Очистить память:
  Marked.Free;         // очередь вершин для маркировки
  Tree.ClrAndDestroy; // остовное дерево
  Tree.Free;
  Que.Free;            // очередь линков
end;


// GenHamPath_Full -- Поиск гамильтоновой цепочки полным перебором

function TGraph.GenHamPath_Full(aStart, aFin: TNode;
                                var aCost: integer): TBuffer;
var
    Cost: integer;    // текущая стоимость контура
    Count: integer;   // счётчик окрашенных вершин
    Node: TNode;      // текущая вершина
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Процедура сохранения текущего решения.
  // Вызывается после окраски всех вершин.
  // aFin - последняя окрашенная вершина
  procedure SaveResult;
  begin
    aCost:= Cost;          // сохраняем стоимость контура
    Result.Clear;          // очищаем буфер результата
    Node:= aFin;           // заполняем буфер с последней вершины
    repeat
      Result.Push(Node);   // заносим в буфер в обратном порядке
      Node:= Node.mPred;   // предыдущая вершина
    until Node=aStart;     // пока не достигнем начальной вершины
    Result.Push(Node);     // заносим конечную вершину
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Рекурсивная процедура поиска в глубину

  procedure Local(aNode: TNode);

  var L : TLink;  // текущий исходящий линк
  begin { Local }
    aNode.mColor:= CBlack;  // красим данную вершину
    Inc(Count);             // и наращиваем счётчик окрашенных
    // Все вершины (кроме последней) окрашены?
    if Count < mNodes.GetCount-1 then begin
      // Нет, ищем соседнюю неокрашенную перебором исходящих линков
      L:= aNode.OutLinkFirst;
      while Assigned(L) do begin
        if (L.mDest <> aFin) and (L.mDest.mColor = CWhite) then begin
          // Нашли неокрашенную и не последнюю:
          Inc(Cost, L.mValue);         // накапливаем стоимость
          if Cost < aCost then begin  // если стоимость ниже минимальной
            L.mDest.mPred:= aNode;    // то метим предыдущей вершиной
            Local(L.mDest);           // и рекурсивно вызываем эту же проц.
          end;
          Dec(Cost, L.mValue);         // восстанавливаем стоимость
        end;
        L:= aNode.OutLinkNext;         // следующий исходящий линк
      end;
    end else begin
      // Здесь Count = mNodes.GetCount-1 -- все вершины, кроме aFin, окрашены
      // Ищем замыкающий линк на вершину aFin
      L:= aNode.OutLinkFirst;
      while Assigned(L) and (L.mDest<>aFin)
```

```
        do L:= aNode.OutLinkNext;
      if Assigned(L) then begin
        // Замыкающий линк найден:
        Inc(Cost, L.mValue);       // накапливаем стоимость
        L.mDest.mPred:= aNode;     // метим предыдущей вершиной
        if Cost < aCost            // если она меньше минимальной
          then SaveResult;         // сохраняем контур и его стоимость
        Dec(Cost, L.mValue);       // восстанавливаем стоимость
      end;
    end;
    // При выходе восстанавливаем счётчик вершин и цвет
    Dec(Count);
    aNode.mColor:= CWhite;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

begin { TGraph.GenHamPath_Full }

  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;               // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  aCost:= MaxInt;// текущая минимальная стоимость = Maxint
  ResetNodes;           // очистка вспомогательных полей
  Cost:= 0;             // накопленная стоимость = 0
  Count:= 0;            // счётчик окрашенных вершин = 0
  Local(aStart);        // вызов рекурсивной процедуры
  // Если путь не обнаружен, возвращаем минус 1
  if aCost = MaxInt then aCost:=-1;
end;


// GenHamPath_Greed -- Поиск гамильтоновой цепочки
// либо жадным методом (aGreed=true)
// либо случайным перебором (aGreed=false);

function TGraph.GenHamPath_Greed(aStart, aFin: TNode;
                                 var aCost: integer;
                                 aGreed: boolean): TBuffer;

  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение первого встретившегося белого линка
  function GetAnyLink(aNode: TNode): TLink;
  begin
    Result:= aNode.OutLinkFirst;
    while Assigned(Result) and
         (Result.mDest.mColor <> CWhite)
      do Result:= aNode.OutLinkNext;
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Извлечение ближайшего белого линка
  function GetBestLink(aNode: TNode): TLink;
  var  Link: TLink;
       BestCost: integer;
  begin
    Result:= nil;
    BestCost:= MaxInt;
    Link:= aNode.OutLinkFirst;
    while Assigned(Link) do begin
      if (Link.mDest.mColor = CWhite) and
         (Link.mValue < BestCost) then begin
        BestCost:= Link.mValue;
```

```
        Result:= Link;
      end;
      Link:= aNode.OutLinkNext;
    end; // while
  end;
  //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
var
    Cost: integer;    // текущая стоимость цепи
    Count: integer;   // счётчик неокрашенных вершин
    Node: TNode;      // текущая вершина
    L : TLink;        // текущий линк

begin { GenHamPath_Greed }
  Result:= TBuffer.Create;  // создаём пустой буфер
  aCost:= -1;                 // на случай отсутствия решения
  // Если граф не сильно связан, то решения нет:
  if not IsLinked then Exit;
  ResetNodes;           // очистка вспомогательных полей
  aFin.mColor:= CBlack; // красим конечную вершину
  Cost:= 0;             // накопленная стоимость = 0

  Count:= mNodes.GetCount-2; // счётчик неокрашенных вершин
  Node:= aStart;
  while Count > 0 do begin
    Node.mColor:= CBlack;
    Result.Put(Node);
    // перебираем соседние неокрашенные вершины
    if aGreed
      then L:= GetBestLink(Node) // ищем ближайшую неокрашенную
      else L:= GetAnyLink(Node); // ищем любую неокрашенную
    // Если неокрашенных нет, прервать цикл
    if not Assigned(L) then break;
    // Нашли неокрашенную:
    Inc(Cost, L.mValue);    // накапливаем стоимость
    Dec(Count);             // счётчик - 1
    Node:= L.mDest;         // следующая вершина
  end; // while

  if Count=0 then begin
    // Здесь пройдены все вершины, кроме последней
    Result.Put(Node);       // заносим в результат предпоследнюю
    L:= Node.GetLink(aFin); // линк из предпоследней в последнюю
    if Assigned(L) then begin
      Result.Put(aFin);
      Inc(Cost, L.mValue);  // накапливаем стоимость
      aCost:= Cost;         // возвращаем стоимость
    end else Result.Clear;  // если нет завершающего линка
  end;
end;
```

```
//#########################################################################
//                                                                        #
// THugeGraph -- Огромный граф (узлы содержат числа типа Integer)         #
// Применяется как вспомогательный в других методах и при тестировании    #
//                                                                        #
//#########################################################################


// Конструктор случайного графа

constructor THugeGraph.GenRandom(aDir: boolean; // направленность
                      aLoadNodes, aLoadLinks,   // нагруженность
                      aNodes,                   // количество вершин
                      aLinks                    // плотность связей,%
                      : integer);
const CArr : array[boolean] of char = ('F','T');

var     nm: string;
         i: integer;
     Links: integer;
     nS,nD: integer;
      Link: TLink;

begin
  // Формируем имя графа:
  nm:='Huge: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(aNodes)+' : '+
         IntToStr(aLinks);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 1 to aNodes do begin
    InsertNode(TNodeInt.Create(i, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи:
  Links:= aNodes*(aNodes-1);  // предельное количество связей
  if aLinks < 100 {%}
    then Links:= Round(Links * aLinks / 100);

  while Links > 0 do begin
    nS:= 1+Random(aNodes);
    nD:= 1+Random(aNodes);
    if nS = nD then Continue;
    Link:= GetLinkByNumber(nS, nD);
    if not Assigned(Link) then begin
      MakeLink(nS, nD, 1+Random(aLoadLinks));
      Dec(Links);
      if not mDirect then Dec(Links);
    end;
  end;
end;

// Конструктор полного двудольного графа

constructor THugeGraph.GenDicoty(aLoadLinks, aPairs: integer);
var nm : string; // имя графа
     n : integer;
```

```
    N1, N2 : TNode;

begin
  // Формируем имя графа:
  nm:='Huge : Dicoty '+ IntToStr(aLoadLinks)+ ' : '+ IntToStr(2*aPairs);
  // Создаём пустой граф:
  inherited Create (nm,      // имя
                    true,    // орграф
                    true,    // вершины нагружены
                    true);   // дуги нагружены
  // создаём и вставляем вершины с номером и значением n
  for n:= 1 to 2*aPairs do begin
    mNodes.Insert(TNodeInt.Create(n, n, Self));
  end;
  // Создаём дуги из всех нечётных (левых) вершин во все чётные (правые)
  N1:= NodeFirst;
  while Assigned(N1) do begin
    if (N1.mValue mod 2) = 1 then begin
      PosPush;
      N2:= NodeFirst;
      while Assigned(N2) do begin
        if (N2.mValue mod 2) = 0
          then SetLink(N1, N2, 1+Random(aLoadLinks));
        N2:= NodeNext;
      end;
      PosPop;
    end;
    N1:= NodeNext;
  end;
end;


// Конструктор полного графа

constructor THugeGraph.GenFull(aDir: boolean; // направленность
                    aLoadNodes, aLoadLinks,   // нагруженность
                    aNodes                    // количество вершин
                    : integer);

const CArr : array[boolean] of char = ('F','T');
var i : integer;
    N1, N2 : TNode;
    nm : string;
begin
  // Формируем имя графа:
  nm:='Huge Full: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(aNodes);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 1 to aNodes do begin
    mNodes.Insert(TNodeInt.Create(i, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи между всеми вершинами графа
  N1:= NodeFirst;
  while Assigned(N1) do begin
    PosPush;
```

```
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N1 <> N2 then SetLink(N1, N2, 1+Random(aLoadLinks));
      N2:= NodeNext;
    end;
    PosPop;
    N1:= NodeNext;
  end;
end;


// Получение указателя на связь по номерам исходящей и входящей вершин

function THugeGraph.GetLinkByNumber(aSource, aDest: integer): TLink;
var S: TNodeInt;
begin
  Result:= nil;
  S:= GetNode(aSource);
  if not Assigned(S) then Exit;
  Result:= S.GetLinkByNumber(aDest);
end;


// Получение указателя на вершину по её номеру

function THugeGraph.GetNode(aNumber: integer): TNodeInt;
begin
  mNodes.PositionPush;
  Result:= TNodeInt(mNodes.GetFirst);
  while Assigned(Result) do begin
    if Result.mNumber = aNumber then Break;
    Result:= mNodes.GetNext as TNodeInt;
  end;
  mNodes.PositionPop;
end;


// Установка связи по номерам исходящей и входящей вершин

procedure THugeGraph.MakeLink(aSource, aDest, aVal: integer);
var S, D: TNode;
begin
  if GetLinkByNumber(aSource, aDest) <> nil then Exit;
  S:= GetNode(aSource);
  D:= GetNode(aDest);
  if Assigned(S) and Assigned(D)
    then SetLink(S, D, aVal)
end;
```

```
// Функция для создания огромного графа с заданными параметрами

function GenHugeGraph (aDir: boolean;           // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;
begin
  if aLinks >= 100 {%}
    then Result:= THugeGraph.GenFull(aDir,          // направленность
                             aLoadNodes, aLoadLinks, // нагруженность
                             aNodes                  // количество вершин
                             )
    else Result:= THugeGraph.GenRandom(aDir,        // направленность
                             aLoadNodes, aLoadLinks, // нагруженность
                             aNodes,                 // количество вершин
                             aLinks                  // плотность связей,%
                             );
end;


end.
```

Приложение H
Модуль GrChars

```
unit GrChars;
{$I Common.inc}
//*************************************************************************
//    Модуль генрации графов, вершины которого обозначены символами      *
//                                                                       *
// Содержит объекты:                                                     *
// TNodeChar -- вершина-символ                                           *
// TGraphChars  -- Граф, сотавленный из вершин-символов                  *
//*************************************************************************

interface

uses Root, Graph;

type

  // TNodeChar -- вершина-символ
  TNodeChar = class (TNode)
    private
      function GetLinkByName(aName : char): TLink;
    public
      mName  : char;  // хранимый символ
      constructor Create(aName: char; aVal: integer; aOwner: TGraph);
      function  Copy: TItem; override;
      function  GetName: string;  override;
  end;

  // TGraphChars  -- Граф, сотавленный из вершин-символов
  TGraphChars = class (TGraph)
    private
      function  GetLinkByName(aSource, aDest : char): TLink;
      procedure MakeLink(aSource, aDest : char; aVal : integer);
    public
      constructor Load(const aName: String);
      constructor GenRandom(aDir: boolean;
                            aLoadNodes, aLoadLinks,
                            aNodes, aLinks : integer);
      constructor GenFull(aDir: boolean;
                          aLoadNodes, aLoadLinks, aNodes : integer);
      procedure LoadFlowData(const aName: String);
      procedure Save(const aName: String); override;
      function  GetNode(aName : char): TNodeChar;
  end;

// Генерация либо полного, либо неполного случайных графов

function GenCharsGraph(aDir: boolean;          // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                 // количество вершин
                       aLinks                  // плотность связей,%
                       : integer): TGraph;

////////////////////////////////////////////////////////////////////////////
implementation
uses SysUtils;
////////////////////////////////////////////////////////////////////////////
```

```
// Генерация либо полного, либо неполного случайных графов

function GenCharsGraph(aDir: boolean;           // направленность
                       aLoadNodes, aLoadLinks, // нагруженность
                       aNodes,                  // количество вершин
                       aLinks                   // плотность связей,%
                       : integer): TGraph;
begin
  if aLinks >= 100 {%}
    then Result:= TGraphChars.GenFull(aDir,          // направленность
                              aLoadNodes, aLoadLinks, // нагруженность
                              aNodes                  // количество вершин
                              )
    else Result:= TGraphChars.GenRandom(aDir,        // направленность
                              aLoadNodes, aLoadLinks, // нагруженность
                              aNodes,                 // количество вершин
                              aLinks                  // плотность связей,%
                              );
end;

////////////////////////////////////////////////////////////////////////////
// TNodeChar -- вершина-символ
////////////////////////////////////////////////////////////////////////////

constructor TNodeChar.Create(aName: char; aVal: integer; aOwner: TGraph);
begin
  inherited Create(aVal, aOwner);
  mName:= aName;
end;


function TNodeChar.Copy: TItem;
begin
  Result:= TNodeChar.Create(mName, mValue, mOwner);
end;

// Поиск линка по имени смежной вершины

function TNodeChar.GetLinkByName(aName: char): TLink;
begin
  OutPosPush;
  Result:= OutLinkFirst;
  while Assigned(Result) do begin
    if (Result.mDest as TNodeChar).mName = aName then Break;
    Result:= OutLinkNext;
  end;
  OutPosPop;
end;

function TNodeChar.GetName: string;
begin
  Result:= mName;  // возвращаем символ - название вершины
end;

////////////////////////////////////////////////////////////////////////////
// TGraphChars -- граф с вершинами-символами
////////////////////////////////////////////////////////////////////////////

// Конструктор полного графа

constructor TGraphChars.GenFull(aDir: boolean;
                                aLoadNodes, aLoadLinks, aNodes: integer);
```

```
const CArr : array[boolean] of char = ('F','T');
var i : integer;
    N1, N2 : TNode;
    name: Char;
    nm : string;
begin
  // Формируем имя графа:
  nm:='Full: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(aNodes);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 0 to aNodes-1 do begin
    name:= Char(Ord('A')+i);
    InsertNode(TNodeChar.Create(name, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи между всеми вершинами графа
  N1:= NodeFirst;
  while Assigned(N1) do begin
    PosPush;
    N2:= NodeFirst;
    while Assigned(N2) do begin
      if N1 <> N2 then SetLink(N1, N2, 1+Random(aLoadLinks));
      N2:= NodeNext;
    end;
    PosPop;
    N1:= NodeNext;
  end;
end;

// Конструктор случайного графа
// Если aNodes < 0, то число вершин точное, а иначе случайное
// Если aLinks < 0, то плотность графа (%) точная, а иначе случайная

constructor TGraphChars.GenRandom(
                            aDir: boolean;  // признак орграфа
                            aLoadNodes,     // нагруженность вершин
                            aLoadLinks,     // нагруженность связей
                            aNodes,         // предельное число вершин
                            aLinks:         // предельное число связей, %
                            integer);

const CArr : array[boolean] of char = ('F','T');

var Nodes, Links : integer;
    i, Limit : integer;
    Link: TLink;
    name, nS, nD: Char;
    nm : string;

begin
  if aNodes > 0
    then  // кол-во вершин случайное
      Nodes:= 2 + aNodes div 2 + Random(aNodes div 2)
    else  // кол-во вершин точное
      Nodes:= Abs(aNodes);
  Limit:= Nodes*(Nodes-1) div 2;  // предельное количество связей
```

```
  // кол-во связей
  if aLinks > 0 then begin
    // Количество связей случайно
    Links:= aLinks;
    if Links > 100 then Links:= 100;
    Links:= Limit div 4 + Random((aLinks*Limit) div 100);
  end else begin
    // Количество связей точно
    Links:= Abs(aLinks);
    if Links > 100 then Links:= 100;
    Links:= (Links*Limit) div 100;
  end;

  // Формируем имя графа:
  nm:='Random: '+ CArr[aDir] +' : '+
         IntToStr(aLoadNodes)+ ' : '+
         IntToStr(aLoadLinks)+ ' : '+
         IntToStr(Nodes)+' : '+
         IntToStr(Links);

  // Создаём пустой граф:
  inherited Create (nm, aDir, aLoadNodes > 0, aLoadLinks > 0);

  // создаём вершины
  for i:= 0 to Nodes-1 do begin
    name:= Char(Ord('A')+i);
    InsertNode(TNodeChar.Create(name, 1+Random(aLoadNodes), Self));
  end;

  // Создаём связи
  repeat
    nS:= Char(Ord('A')+Random(Nodes));
    nD:= Char(Ord('A')+Random(Nodes));
    if nS = nD then Continue;
    Link:= GetLinkByName(nS, nD);
    if not Assigned(Link) then begin
      MakeLink(nS, nD, 1+Random(aLoadLinks));
      Dec(Links);
    end;
  until Links <= 0;
end;

{                Структура взвешенного графа
Gragh - 2
0 - граф(0), орграф(1)
1 - нагруженность вершин
1 - нагруженность рёбер (дуг)
3 - количество вершин
A=1 B=3 C=2
A -> B=5 C=7
B -> A=5 C=6
C -> A=7 B=6
}

constructor TGraphChars.Load(const aName: String);
const CharsSet = ['a'..'z', 'A'..'Z'];
var F: TextFile;
    nodes, n, val : integer;
    nm : string;        // имя графа
    name : char;        // имя узла = 'A'..'Z'
    Dir, LN, LL : boolean;
    S, D : char;        // имя источника и приёмника связи
```

```
    node : TNodeChar;  // очередной узел
    err : boolean;     // признак ошибки
begin
  if not SysUtils.FileExists(aName) then begin
    Fail;
    Exit;
  end;
  err:= false; n:= 0;
  AssignFile(F, aName); Reset(F);

  if not Eof(F) then Readln(F, nm) else err:= true;
  if not Eof(F) then Readln(F, n) else err:= true;
  Dir:= n<>0;   // признак орграфа
  if not Eof(F) then Readln(F, n) else err:= true;
  LN:= n<>0;    // признак нагруженных вершин
  if not Eof(F) then Readln(F, n) else err:= true;
  LL:= n<>0;    // признак нагруженных связей

  inherited Create(nm, Dir, LN, LL);  // создание пустого графа

  // Количество вершин:
  if not Eof(F) then Readln(F, n) else err:= true;
  nodes:= n;

  // Чтение вершин
  while not err and not Eof(F) and (n>0) do begin
    name:= '0'; val:= 1;
    Dec(n);
    // чтение имени (a..z, A..Z)
    while not Eoln(F) do begin
      Read(F, name);
      if name in CharsSet then Break;
    end;
    // чтение веса вершины
    if mLoadNodes then begin
      Read(F, S);     // пропускаем знак =
      Read(F, val);   // чтение веса
    end;
    node:= TNodeChar.Create(name, val, Self);
    if not InsertNode(node)
      then node.Free;  // если пытались повторно, удаляем дубликат
  end;
  Readln(F);

  if n>0 then err:= true;  // не все узлы прочитаны

  // Чтение связей
  while not err and not Eof(F) do begin
    val:= 1;
    S:='0'; D:='0';
    // чтение вершины-источника
    while not Eoln(F) do begin
      Read(F, S);
      if S in CharsSet then Break;
    end;
    // чтение вершин-приёмников
    while not Eoln(F) do begin
      // чтение имени
      while not Eoln(F) do begin
        Read(F, name);
        D:= name;
        if D in CharsSet then Break;
```

```
      end;
      if not (D in CharsSet) then Break;
      // Для нагруженных связей:
      if mLoadLinks then begin
         Read(F, name);    // пропускаем знак =
         Read(F, val);     // чтение веса
      end;
      MakeLink(S, D, val);
    end;  // while
    Readln(F);
    Dec(nodes);
    if nodes=0 then Break;  // если все вершины
  end;  // while
  Close(F);

  if err then begin
    if Assigned(mNodes) then begin
      mNodes.ClrAndDestroy;
      mNodes.Free;
    end;
    Fail;
  end;
end;

// Чтение данных к задачам на потоках
// Формат входных данных:
{
FLOW: -- стартовый маркер
XY = mL mH  -- дуга, минимальный поток, максимальный поток
...
}
procedure TGraphChars.LoadFlowData(const aName: String);
var F: TextFile;
    S: string;
    C1, C2: char;
    Low, High : integer;
    L: TLink;
begin
  AssignFile(F, aName); Reset(F);
  // Поиск стартового маркера
  while not Eof(F) do begin
    Readln(F,S);
    if Pos('FLOW',S)<>0 then Break;
  end;
  while not Eof(F) do begin
    // C1, C2 -- имена источника и приёмника дуги
    C1:='-'; C2:='-';
    if not Eoln(F) then Read(F, C1) else begin Readln(F); Continue end;
    if not Eoln(F) then Read(F, C2) else begin Readln(F); Continue end;
    L:= GetLinkByName(C1, C2);
    if Assigned(L) then begin
      while not Eoln(F) do begin
        Read(F,C1);
        if C1='=' then Break;
      end;
      if not Eoln(F) then Read(F, Low) else begin Readln(F); Continue end;
      if not Eoln(F) then Read(F, High) else begin Readln(F); Continue end;
      // Сохраняем данные о потоке
      L.mLow:= Low;
      L.mHigh:= High;
    end;
    Readln(F);
```

```
  end;
  Close(F);
end;

// Сохранение графа в текстовом файле

procedure TGraphChars.Save(const aName: String);
var F: TextFile;
    i, j: integer;
    N: TNodeChar;
    L: TLink;

begin
  AssignFile(F, aName);
  if FileExists(aName) then Append(F) else Rewrite(F);
  Writeln(F, mName);
  Writeln(F, Ord(mDirect), ' - тип графа (1 = орграф)');
  Writeln(F, Ord(mLoadNodes), ' - вершины (1 = нагруженные)');
  Writeln(F, Ord(mLoadLinks), ' - дуги (1 = нагруженные)');
  Writeln(F, mNodes.GetCount, ' - количество вершин');

  for i:= 1 to mNodes.GetCount do begin
    N:= mNodes.GetItem(i) as TNodeChar;
    if mLoadNodes
      then Write(F, N.mName+'=', N.mValue, ' ')
      else Write(F, N.mName, ' ');
  end;
  Writeln(F);

  for i:= 1 to mNodes.GetCount do begin
    N:= TNodeChar(mNodes.GetItem(i));
    Write(F, N.mName,' -> ');
    for j:= 1 to N.OutGetCnt do begin
      L:= N.OutGetLink(j);
      if N.mOwner.mLoadLinks
        then Write(F, (L.mDest as TNodeChar).mName, '=', L.mValue:2,' ')
        else Write(F, (L.mDest as TNodeChar).mName,' ');
    end;
    Writeln(F);
  end;
  Close(F);
end;

// Получение указателя на связь по именам исходящей и входящей вершин

function TGraphChars.GetLinkByName(aSource, aDest: char): TLink;
var S: TNodeChar;
begin
  Result:= nil;
  S:= GetNode(aSource);
  if not Assigned(S) then Exit;
  Result:= S.GetLinkByName(aDest);
end;

// Получение указателя на вершину по её имени

function TGraphChars.GetNode(aName: char): TNodeChar;
begin
  mNodes.PositionPush;
  Result:= TNodeChar(mNodes.GetFirst);
  while Assigned(Result) do begin
    if Result.mName = aName then Break;
```

```
    Result:= mNodes.GetNext as TNodeChar;
  end;
  mNodes.PositionPop;
end;

// Установка связи по именам исходящей и входящей вершин

procedure TGraphChars.MakeLink(aSource, aDest : char; aVal: integer);
var S, D: TNode;
begin
  if GetLinkByName(aSource, aDest) <> nil then Exit;
  S:= GetNode(aSource);
  D:= GetNode(aDest);
  if Assigned(S) and Assigned(D)
    then SetLink(S, D, aVal)
end;


end.
```