МИНОБРНАУКИ РОССИИ

–––––––——————————–––––––

Санкт-Петербургский государственный  
электротехнический университет «ЛЭТИ»

————————————————————

##### **П. Г. Колинько**

##### **АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ**

**Часть 1. Пользовательские структуры данных**

Конспект лекций

Выпуск 2201

Санкт-Петербург  
СПбГЭТУ «ЛЭТИ»  
2021

УДК 004.424:004.422.63(075.8)

Колинько П. Г. Алгоритмы и структуры данных. Часть 1. Пользовательские структуры данных: Конспект лекций. Вып. 2201. –– СПб.: СПбГЭТУ «ЛЭТИ», 2021. — 529 с.

Излагается содержание первой части двухсеместрового курса.

Пособие предназначено для студентов бакалавриата по направлению 230100.62 «Информатика и вычислительная техника» заочной формы обучения, а также для дополнительного чтения студентам очной и очно-заочной форм.

Одобрено  
Методической комиссией факультета   
информатики и вычислительной техники   
СПбГЭТУ «ЛЭТИ»  
в качестве учебного пособия

© П. Г. Колинько, 2022 (

# 1. Цели и задачи курса

Колинько П. Г. Алгоритмы и структуры данных. Часть 1. Пользовательские структуры данных

Курс «Алгоритмы и структуры данных» задуман как естественное продолжение курса «Основы программирования», переход от изучения возможностей и конструкций алгоритмического языка к проектированию программ, реализующих сложные алгоритмы. В курсе изучается связь свойств программного изделия с рациональностью организации данных.

Структура — это пользовательский тип данных, с которым связана некоторая область памяти и набор допустимых операций с ней, т. е. объект. Работа со структурами данных — это объектно-ориентированное программирование.

В качестве базового языка программирования выбран язык С++ в варианте последних стандартов С++11, С++14, С++17, С++20. Курс начинается с изучения основ С++ и заканчивается (во второй части курса) знакомством со способами взаимодействия в программе нескольких классов (наследованием и т. п.), обработкой исключительных ситуаций и ис­поль­зова­ни­ем Стандартной библиотеки шаблонов (STL).

В качестве источника алгоритмов используется параллельно изучаемый курс «Дискретная математика и теория алгоритмов». Из этого курса заимствуются постановки задач и доказательства работоспособности алгоритмов, а в настоящем курсе делается упор на их реализацию в виде программы на ЭВМ, тестирование и отладку.

Поэтому в качестве литературы по курсу «Алгоритмы и структуры данных» студентам предлагаются — в качестве источника сведений об алгоритмах — книги по дискретной математике. Это, в первую очередь:

— Поздняков С. Н., Рыбин С. В. Дискретная математика: учебник для вузов. М.: Академия, 2008. – 448 с.

— Хагерти Р. Дискретная математика для программистов. Изд. 2‑е, испр. — М.: Техносфера, 2012. — 400 с.;

— Новиков Ф. А. Дискретная математика для программистов. — СПб.: Питер, 2000. — 304 с., ил.

Многие идеи курса взяты из классических источников:

— Ахо Дж., Хопкрофт А., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979;

— Ахо Дж., Хопкрофт А., Ульман Дж. Структуры данных и алгоритмы. — СПб.: И. Д. Вильямс, 2001. — 382 c.;

— Липский В. Комбинаторика для программистов. — М.: Мир, 1978. — 213 с.;

— Макконелл Дж. Основы современных алгоритмов. 2-е. изд. — М.: Техносфера, 2004. — 368 с.

— Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. — М.: Мир, 2013.

В качестве дополнительных источников информации об алгоритмах могут быть полезны книги:

— Гэри М., Джонсон Д. Вычислительные машины и трудно решаемые задачи. — М.: Мир, 1982. — 419 с.;

— Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е изд. / пер. с англ. — М.: Мир, 2005. — 1296 с.: ил.;

— Седжвик Р. Фундаментальные алгоритмы С++. — К.: Диасофт, 2001. — 484 с.;

— Седжвик Р. Алгоритмы на С++ / пер. с англ. — М.: И. Д. Вильямс, 2011. — 1156 с.: ил.

Другой очень важный источник информации по курсу — книги по программированию. И здесь, при кажущемся изобилии, действительно полезных книг очень немного.

Есть книги по программированию на «суржике», т. е. на Си с элементами С++. Это в принципе неправильно, С++ задуман не как дополнение, а как замена языка Си с его особенностью слишком многое оставлять «на усмотрение программиста» и провоцировать его на трюки, создавать проблему неинициализированных и повторно используемых переменных и злоупотребления указателями.

Есть много книг с добросовестным описанием языка (копия стандарта), описывающие «как писать», но не говорящие «зачем». Есть книги, переучивающие готовых программистов с других языков. Так, классическая книга по языку Си Кернигана и Ритчи переучивает программистов с фортрана. Есть книги, натаскивающие на конкретные системы программирования, сейчас уже устаревшие — Borland C++, Visual C++ 2005, 2008.

Так, великолепный учебник Лафоре (Лафоре Р. Объектно-ориентированное программирование в С++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2015. — 928 с.: ил.) ориентирован на уже не актуальный Borland C++.

По С++ наиболее полезны книги автора языка С++ Б. Страуструпа, в которых говорится не только «как писать», но и «зачем», откуда появилось то или иное средство языка и какие задачи он призвано решать:

— Страуструп Б. Язык программирования С++. 2‑е доп. изд. — М.: Бином-пресс, 2001. – 1098 с.;

— Страуструп Б. Язык программирования С++. Специальное издание. Пер, с англ. — М.: Изд-во Бином, 2015. — 1136 с.: ил.

Общий недостаток всех перечисленных книг: в них ничего нет о новом стандарте языка — С++11, снявшем многие из существовавших в языке проблем и добавившем многие возможности. Зачем это понадобилось, можно узнать из книги Дьюхэрста:

— Стефан К. Дьюхэрст. Скользкие места С++. Как избежать проблем при проектировании и компиляции ваших программ. — М., ДМК Пресс, 2012. — 264 с., ил.

В этой книге — много о том, как обычно пишут программы на С++ и как это делать правильно, и говорится о проблемах, которые решены новым стандартом языка. Книга написана до появления этого стандарта. К сожалению, более позднее издание этой книги (2017 г.) оказалось стереотипным.

О проблемах при работе с языком С++ и рекомендациях по решению сложных задач, тонкостям применения языка можно прочесть в книгах серии «C++ in Depth», написанных Скоттом Мейерсом, Гербом Саттером, Андреем Александреску и др.:

— Мейерс С. Наиболее эффективное использование С++. 35 новых рекомендаций по улучшению ваших программ и проектов. — М.: ДМК Пресс, 2012. — 294 с.: ил.;

— Саттер Г. Решение сложных задач на С++. Пер. с англ. — М.: И. Д. Вильямс, 2015. — 400 с.: ил.

Лучшая из существующих на сегодня книг, учитывающих новый стандарт языка — это:

— Прата С. Язык программирования C++. 6‑е изд. — М.: И. Д. Вильямс, 2011. — 1244 с.

Она интересна тем, что в ней параллельно описываются языки Си, С++ (в стандарте 97/03) и С++11, показаны все различия в коде.

На новый стандарт ориентирован популярный учебник С++:

— Липпман С. Б., Лажойе Ж., Му Б. Э. Язык программирования С++. Базовый курс. 5-е изд. Пер. с англ. — М.: И.Д.Вильямс, 2014. — 1120 с.: ил.

Новая книга Страуструпа учитывает стандарты С++11 и   
С++14:

— Страуструп Б. Программирование: принципы и практика с использованием С++: Второе издание. — СПб.: ООО «Диалектика», 2019. — 1328 с.

Язык С++ продолжает развиваться, уже доступны компиляторы для стандарта С++14, учтённого в книгах Питера Готтлинга и Скотта Мейерса:

— Готтшлинг П. Современный С++ для программистов, инженеров и учёных. — М.: И. Д. Вильямс, 2016. — 512 с.: ил.;

— Мейерс С. Эффективный и современный С++: 42 рекомендации по использованию С++11 и С++14: пер. с англ. — М.: И. Д. Вильямс, 2018. — 304 с.: ил.

В качестве справочника по применению Стандартной библиотеки шаблонов (STL) рекомендуется книга:

— Джоссатис Н. М. Стандартная библиотека С++: справочное руководство. 2-е изд. Пер. с англ. — М.: И. Д. Вильямс, 2014. — 1136 с.: ил.

Большинство упомянутых книг доступно на машинных носителях. Их можно скачать из интернета, в частности, с популярного сайта http://proklondike.com.

Актуальный справочник по функциям и другим средствам языка С++, включая последние стандарты: http://ru.cppreference.com

Поскольку русификация этого справочника сделана с помощью машинного перевода, иногда может быть полезен и оригинал:

http://en.cppreference.com

# 2. Оценка временной сложности алгоритмов

При проектировании алгоритмов, как правило, не представляет интереса точное число шагов, необходимых для решения задачи на конкретном наборе данных. Гораздо важнее знать, как будет изменяться время решения задачи *T*, если размер входа *n* растёт.

Класс алгоритмов, время работы которых растёт, по крайней мере, так же быстро, как некоторая функция *f*(*n*), обозначается как Ω( *f*(*n*)). Это означает, что при всех *n*, превышающих порог *n*0, *T*(*n*) ≥ *C*.*f*(*n*) для некоторого положительного числа *C*. Оценка времени работы снизу может представлять интерес только как теоретическая нижняя граница эффективности любого алгоритма для некоторой задачи, которую преодолеть невозможно.

Класс алгоритмов, время работы которых растёт не быстрее функции *f*(*n*), обозначается **O( *f*(*n*)**), что означает существование положительных чисел *n*0 и *C* таких, что при *n* > *n*0 *T*(*n*) ≤ *C*.*f*(*n*). Этот класс — важнейшая характеристика алгоритма, его *временная сложность*. По скорости роста этого времени в зависимости от размера входа алгоритмы делятся на следующие классы временной сложности:

— алгоритмы константной сложности — *T*(*n*) ∈ O(1);

— логарифмической сложности — *T*(*n*) ∈ O( log *n*);

— линейной сложности — *T*(*n*) ∈ O(*n*);

— квадратичной сложности — *T*(*n*) ∈ O(*n*2);

— кубической сложности — *T*(*n*) ∈ O(*n*3);

— полиномиальной сложности — *T*(*n*) ∈ O(*nk*), где *k* = const; *k* = 0, 1, 2 или 3 — это частные случаи классов полиномиальной сложности;

— экспоненциальной сложности — *T*(*n*) ∈ O(*an*).

Очевидно, что классы в этом перечне упорядочены по возрастанию мощности. Так, класс O(1) является подмножеством любого из остальных классов. Задача программиста — найти или разработать алгоритм класса минимально возможной мощности и реализовать его так, чтобы оценка временной сложности не ухудшилась.

Алгоритм, для которого оценки Ω( *f*(*n*)) и O( *f*(*n*)) совпадают, называется *оптимальным*. Так, очевидно, что алгоритм, имеющий на входе некоторое множество, будет оптимальным, если его временная сложность O(1). Такой алгоритм можно попытаться найти, если задача не требует рассмотреть множество целиком. Если же требуется что-то сделать с каждым элементом множества мощностью *n*, оптимальный алгоритм будет иметь сложность O(*n*). Если имеется два множества, и нужно обработать все возможные пары их элементов, можно ожидать сложности O(*n*2), для трёх множеств, если обрабатываются все тройки, — O(*n*3), и т. д.

Если же для получения результата необходимо рассмотреть все подмножества исходного множества или все перестановки его элементов — это задача на полный перебор, принадлежащая классу экспоненциальной сложности. В таких случаях говорят об отсутствии *эффективного* алгоритма.

Время работы программы часто зависит не только от мощности входных данных, но и от того, какие именно данные поступили на вход. В таких случаях делаются две оценки временной сложности:

— для самого неудобного набора данных — сложность «в худшем случае»;

— для типового набора данных — сложность «в среднем».

Тривиальные входные данные («лучший случай») обычно интереса не представляют.

Для оценки временной сложности по реализации алгоритма (**по тексту программы**) можно руководствоваться следующими соображениями:

— операции присваивания (копирования) и проверки условия для базовых типов данных выполняются за константное время. Если при этом вызывается функция, сложность шага алгоритма определяется сложностью функции (в операциях с объектами функции могут вызываться неявно);

— сложность алгоритма, состоящего из последовательности шагов, определяется по самому сложному шагу;

— сложность выбора по условию определяется по самой сложной из альтернатив. В порядке исключения можно не принимать во внимание альтернативы, выбираемые очень редко. Можно учесть такие альтернативы, как «худший случай»;

— если какие-то шаги являются внутренней частью цикла с количеством повторений, зависящим от размера входа *n*, в оценку сложности циклического шага добавляется множитель *n*. Если же количество повторений не зависит от *n*, цикл игнорируется, поскольку его можно рассматривать просто как повторение некоторого количества одинаковых шагов алгоритма;

— рекурсия рассматривается как тот же цикл. Её сложность определяется как произведение сложности одного вызова функции на количество вызовов.

*Пример* *1*. Вычислить *b* = (*a* ∈ *A*), где множество *A* мощностью *nA* представлено массивом целых чисел.

*Решение*:

b = false; for (i = 0; !b && (i < nA); i++) b |= (a == A[ i ]);

Временная сложность алгоритма — O( *nA*). Если элемент *a* найден, алгоритм прекращает работу, выполнив от 1 до *nA* шагов. В среднем количество шагов будет *nA* / 2, в худшем случае (*a* ∉ *A*) — *nA*.

*Пример* *2*. Вычислить *C* = *A* ⋂ *B* для множеств, представленных неупорядоченными массивами.

*Решение*: проверяем все возможные пары элементов двух множеств и отбираем совпадения.

for (i = k = 0; i < nA; i++)  
 for (j = 0; j < nB; j++) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];

Проверка на совпадение и присваивание выполняются за константное время, поэтому сложность алгоритма — O(*nA* × *nB*), или O(*n*2), где *n* — средняя мощность множеств.

*Пример* *3*. Вычислить *D* = *A* ⋂ *B* ⋂ *C*.

Очевидное решение

for (int i = 0, k = 0; A[ i ]; ++i)  
 for (int j = 0; B[ j ]; ++j)  
 for (int r = 0; C[ r ]; ++r)  
 if ((A[ i ] == B[ j ]) && (A[ i ] == C[ r ])) D[ k++ ] = A[ i ];

имеет временную сложность O(*n*3), поскольку перебираются все возможные тройки. Однако перебирать все тройки никакой необходимости нет.

Модифицируем алгоритм:

for (int i = 0, k = 0; A [ i ]; ++i)  
 for (int j = 0; B[ j ]; ++j)  
 if (A[ i ] == B[ j ]) for (int r = 0; C[ r ]; ++r)  
 if (A[ i ] == C[ r ]) D[ k++ ] = A[ i ];

В алгоритме по-прежнему три вложенных цикла, но внутренний цикл теперь зависит от условия *A*[ *i* ] == *B*[ *j* ], которое проверяется *n*2 раз, но удовлетворяется не более чем *n* раз, т. е. рассматриваются только такие тройки, у которых первые два элемента совпадают. Проверка *A*[ *i* ] == *C*[ *r*] выполняется, таким образом, не более *n*2 раз, и общая сложность алгоритма — O(*n*2).

*Пример* *4*. Вычислить *C* = *A* ⋂ *B* для множеств, представленных строками символов.

*Решение*:

for (int i = k = 0; i < strlen(A); ++i)  
 for (int j = 0; j < strlen(B); ++j) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];

По аналогии с примером 2 можно оценить сложность как *O*(*n*2). Однако это неверно, так как в обоих циклах по *n* раз вычисляется функция определения длины строки *strlen*( ), которая подсчитывает в строке количество символов до ближайшего нуля перебором, т. е. имеет линейную сложность. Вычисление этой функции — один из шагов внутренней части обоих циклов. Таким образом, внутренняя часть вложенного цикла состоит из трёх шагов, двух константных (проверка и присваивание) и линейного (вычисление функции). С учётом *n* повторений сложность всего цикла — O(*n*2). Внешний цикл добавляет сюда ещё шаг вычисления функции сложностью *O*(*n*). Сложность его внутренней части — O(*n*2), а всего алгоритма — O(*n*3)! Это цена экономии двух целых переменных. На самом деле нужно вычислить пределы заранее *nA* = *strlen*(*A*), *nB* = *strlen*(*B*), а затем использовать алгоритм из примера 2. Альтернатива: если известно, что массивы символов ограничены нулём, это можно использовать, как показано в примере 3. Подумайте, как ограничить нулём результат!

*Контрольные вопросы.*

1. Какой класс временной сложности алгоритма является самым широким?
2. Алгоритм какой временной сложности является самым предпочтительным в общем случае?
3. Алгоритм какой временной сложности является оптимальным для поэлементной обработки множества мощностью *n*?

# 3. Понятие данных. Примитивы и структуры. Стандартные структуры данных

Данные — это то, чем манипулирует программа. Цель работы любой программы состоит в изменении данных. На языке высокого уровня программа манипулирует переменными — именами, каждое из которых связано с некоторой областью памяти, содержимое которой можно изменять, и константами, тоже обозначающими память, но с указанным неизменяемым содержимым. С данными связано понятие типа, который задаёт размер области памяти для переменной (константы) и набор возможных операций с ней.

Мы будем различать данные-примитивы, для обработки которых достаточно одного шага алгоритма, и структуры, требующие нескольких шагов.

Эта классификация относительна. Примитив с точки зрения ЭВМ — это минимальная адресуемая область памяти — байт или машинное слово. С другой стороны, машинное слово состоит из отдельных битов, и существуют средства для работы с битами, т. е. слово — это структура. Мы с вами познакомимся также с ситуациями, когда область данных, состоящая из нескольких частей, копируется в другое место как единое целое, т. е. выступает как примитив. В конце концов, цель создания программы может быть сформулирована как сведение обработки большого объёма сложных данных к одному шагу — запуску программы.

Типы данных, встроенные в язык С++ — это примитивы, в частности: целые (*int*, *long*, *long long*), вещественные (*float*, *double*, *long double*), символьные (*char*), булевские (*bool*), указатели. Для целых и вещественных предусмотрены арифметические операции, для символов — ввод и вывод, для указателей — сложение с целой константой и разыменование для доступа к памяти, адрес которой хранит указатель. С указателем всегда связан тип области памяти, на которую тот указывает. Исключение — указатель на *void* (указатель вообще), который при работе с данными подменяется указателем конкретного типа.

Структуры данных в программе на С++ — это данные, добавляемые в язык пользователем-программистом. Некоторые возможности предусмотрены в самом языке.

В первую очередь это **массивы** — указатели на области памяти, состоящие из заданного количества одинаковых частей — примитивов или структур обязательно одного и того же типа. Например, *int A*[10] обозначает память для 10 целых переменных.

Массив является памятью прямого доступа: любой элемент массива доступен непосредственно, если известен его порядковый номер. Для получения указателя на элемент массива используется индексная арифметика (обычно неявно, с помощью операции «квадратные скобки» [ ] ): *A*[5] — то же, что и \*(*A*+5).

Единственная операция с массивом целиком — это определение его размера: *sizeof*(*A*) вернёт то же, что и 10\**sizeof*(*int*) для указанного выше массива A.

При попытке копирования массива скопируется только указатель на занимаемую им область памяти: в результате *int* \**B* = *A* указатель *B* можно будет индексировать для доступа к данным массива *A*, но операция *sizeof*(*B*) вернёт не размер массива, а размер указателя. Но нет никакого способа определить, указывает ли указатель *B* на массив или на простую переменную, и нет способа определить размер такого массива. Корректность манипуляций с указателем *B* возлагается на программиста.

Объединение данных нескольких, в общем случае разных, типов в единое целое возможно в языке С++ объявлением **структуры**, например:

struct *Type* { int p; char s; int A[10]; };

Объявление структуры — это объявление пользовательского типа данных *Type*. Можно объявлять переменные, массивы и указатели такого типа: *Type* *P*, \**Q*, *R*[15].

Для доступа к полям структуры используется операция «прямое членство», обозначаемая точкой между именем структуры и именем поля, или «косвенное членство — стрелочка между указателем и именем поля:

*P.p*, *Q*->*s*, *R*[0].*A*[5].

В отличие от массивов, структуры можно копировать. Если нужно копировать массив, его можно сделать полем, возможно, единственным, структуры, играющей роль его оболочки. Так, корректно утверждение *R*[0] = *P*, копирующее три поля структуры *R* типа *Type* в нулевой элемент массива структур *R* того же типа.

Важным частным случаем являются структуры *struct*, имеющие в своём составе поля — указатели на структуры того же типа. Из таких структур формируются списки — наборы данных из разных областей памяти, связанных с помощью указателей в единое целое. Элемент списка здесь выступает как примитив, а список в целом — как структура данных.

Объединения (*union*) в общем случае не могут рассматриваться как структуры данных. Они похожи на структуры (*struct*), но каждое поле в них — это отдельный тип для общей для всех полей области данных. Назначение объединений — программистский трюк, предназначенный для обхода некоторых ограничений языка программирования.

Кроме пользовательских, существуют ***абстрактные***, или стандартные структуры данных, с помощью которых описываются алгоритмы на языке математики. Это *множество*, *отображение*, *последовательность*, *очередь* и *стек*.

**Множество** – это совокупность объектов, называемых элементами множеств. Множество может задано перечислением своих элементов:

*S* = { 2, 5, 7, 12, 24 }; *T* = { понедельник, вторник, среда }.

Чтобы показать, что речь идёт о множестве, набор элементов заключается в фигурные скобки. Порядок элементов в перечислении не имеет значения.

Очень важной характеристикой множества является количество элементов в нём — *мощность*. Она может быть бесконечной. В этом случае множество можно задать с помощью предиката — функции на элементах, возвращающей булевское значение (*true* или *false*): *S* = { *x* : *P*(*x*) } — множество всех *x*, для которых *P*(*x*) = *true*. Например, *S* = { *x* : *x* — число месяца, приходящееся на понедельник }.

Некоторые часто используемые множества имеют стандартные названия и обозначения:

∅ — пустое множество;

*U* — полное множество для некоторой задачи, универсум;

*ℕ* = { 1, 2, 3, … } — множество *натуральных* чисел;

*Z* = { 0, ±1, ±2, ±3, … } — множество *целых* чисел;

*R* — множество *вещественных* чисел;

*R+* — множество *неотрицательных вещественных* чисел.

**Отображение** — это некоторая функция *f* на элементах множества *A* со значениями из множества *B*:

f : A → B.

Отображение — это частный случай *бинарного отношения* *R* ⊆ *A* × *B*, удовлетворяющий ограничению: каждый элемент множества *A* связан с единственным элементом множества *B*.

Задать отображение можно, например, перечислением пар:

f = { <a, b>: a ∈ A, b ∈ B }.

Если функция *f* — инъективна (взаимно однозначна), то множества *A* и *B* — изоморфны, и мы можем вместо элементов множества *A* работать с соответствующими элементами множества *B*. Наличие такой функции и определяет изоморфизм

Важный частный случай — использование в качестве множества *B* натуральных чисел: *f* : *A* → *ℕ*. Такое отображение мы будем называть *разметкой* множества A, т. е. присвоением каждому элементу из A порядкового номера для дальнейшей работы с этими номерами вместо самих элементов множества A.

Отображение множества натуральных чисел на произвольное множество *A* образует **последовательность** из элементов этого множества. Задать последовательность можно тоже перечислением элементов множества, но в этом перечислении порядок следования элементов будет важен. Этот факт мы будем обозначать угловыми скобками:

*S* = < 2, 5, 7, 12, 5, 24, 2 >.

В этом примере 5 стоит на 2-й позиции — после 2 и перед 7. В отличие от множества, элементы последовательности в общем случае могут повторяться.

**Очередь и стек** суть последовательности, в которые можно помещать и затем забирать из них элементы множества. В случае очереди элементы множества извлекаются из начала последовательности, а добавляются в её конец. В результате элементы множества будут извлекаться из очереди в том же порядке, в котором в неё помещались. В стеке и добавление, и извлечение элементов выполняются в конце последовательности. Поэтому из стека при извлечении элементов из стека их порядок будет обратный порядку их размещения.

Оценку временной сложности алгоритма, использующего абстрактные структуры данных, мы будем называть теоретической. При реализации алгоритма в виде программы абстрактные структуры данных заменяются данными пользователя-программиста исходя из возможностей, предоставляемых языком С++. В общем случае возможно несколько вариантов такой замены, поэтому ставится задача выбора оптимального варианта и оценка результата такого выбора — оценка временной сложности алгоритма по тексту программы.

Структуры, предоставляемые языком — массив и список — пригодны как основа для реализации любой из перечисленных выше абстрактных структур данных. В конкретную структуру массив превращается, если обрабатывать его по соответствующим правилам. Так, например, если массив должен быть структурой данных для множества, требуется, чтобы тип элемента массива совпадал с типом элемента множества, размер массива был достаточен для хранения множества, были бы определены операции над множествами — объединение, пересечение и т. п. Если в массиве реализован стек, должны быть определены операции «опустить в стек», «поднять из стека».

Программист может сам придерживаться правил, установленных им для конкретной структуры данных и написать необходимые функции обработки в рамках парадигмы процедурного программирования.

Но язык С++ предлагает более удобный механизм — класс, в котором одновременно с набором данных можно определить и набор функций-членов для исключительного доступа к этим данным, получив тем самым полноценный пользовательский тип данных. Конкретные экземпляры данных некоторого класса называются объектами, а программирование с использованием классов — объектно-ориентированным программированием.

Такое программирование является естественной средой для получения навыков работы с пользовательскими структурами данных.

# 4. Множества в памяти ЭВМ

Для представления множества в памяти ЭВМ можно использовать один из двух способов: последовательность из элементов множества или отображение на универсум. Последовательность элементов может храниться в форме массива или списка. Отображение может быть в развёрнутой форме — массив битов (булевских значений) или в компактной — машинное слово.

## 4.1. Представление множества набором элементов

Одним из способов задания множества является простое перечисление входящих в него элементов. В памяти ЭВМ элементы такого множества могут быть расположены в последовательности ячеек, т. е. образуют массив. Это самый экономный способ хранения множества в тех случаях, когда мощность множества известна, а его элементы — данные одного типа (во всех случаях, когда это не так, элементы множества можно расположить в памяти произвольным образом и работать с ними через указатели). Память под массив удобно выделить статически. В этом случае её можно сразу инициализировать.

*Пример*. Объявление и инициализация массива для множества десятичных цифр.

сonst int Nmax = 10;  
char A[ Nmax+1 ] = { '1', '3', '4', '7' };

В память, выделяемую под универсум, помещено множество-константа. Множество символов предполагается обрабатывать как строку, поэтому нужно позаботиться о месте под ограничивающий нуль. Однако в примере выше в этом нет необходимости: поскольку инициализаторов всего 4, остальные элементы массива будут по умолчанию заполнены нулями. Если множество расширяться не будет, размер памяти можно не указывать:

char A[ ] = {"1347"};

Выделяется память под множество из четырёх элементов и ограничивающий нуль.

Мощность множества, заданного строкой-константой, можно вычислять:

int nA = strlen(A);

или просто обрабатывать строку до появления нуля.

Если множество создаётся в результате вычислений, его мощность может быть неизвестна. Поскольку память под массив должна быть выделена до начала работы с ним, приходится делать это с запасом. Проще всего выделить память сразу под универсум, а если универсум слишком велик, то под множество максимальной мощности, которое может быть реально получено. Если же память оказалась исчерпана до окончания вычислений, можно попытаться заказать область памяти большего размера и перенести в неё накопленные элементы множества. Это дорогая операция, поскольку приходится полностью копировать множество. Она может быть невыполнима, если в памяти нет непрерывного участка достаточного размера. Освобождаемый участок памяти тоже не всегда можно использовать. Если же оценка мощности результата слишком пессимистична, память под множество используется нерационально. Так, её приходится выделять даже для результата — пустого множества.

Таких проблем нет, если для представления множества в памяти используется односвязный список. Память в этом случае выделяется под каждый элемент множества отдельно, и её ровно столько, сколько необходимо. Так, под пустое множество-список память вообще не выделяется. Память от удаляемых элементов списка легко использовать под вновь создаваемые элементы. Перенос элемента из одного множества в другое вообще не требует дополнительной памяти.

Главный недостаток способа — необходимость хранить с каждым элементом множества указатель на следующий элемент и тратить время на работу с ним. Так, создание копии списка в другом месте памяти требует не только отдельной обработки каждого элемента множества, но и создания вновь всех указателей. Имеются и скрытые потери: минимальная область, выделяемая в динамической памяти, часто больше, чем требуется для хранения одного элемента множества вместе с указателем.

Реализация алгоритмов для работы с множествами, представленным массивами или списками, отличается только способом перебора этих структур данных. Подтвердим это примером реализации операции принадлежности элемента множеству десятичных цифр.

Для массива используем объявление из предыдущего примера. Множество-список объявим так:

struct Set { char el;  
Set \* next;  
}

Set \*LA; // Указатель на начало списка для множества *A*.

Вариант 1. Поиск элемента множества в строке символов.

bool exist(char s, char A [ ])  
{ bool b{false};  
 for(int i = 0; A[ i ] && !b; ++i ) b |= (s == A[ i ]);  
 return b;  
}

Вариант 2. Поиск элемента множества в списке.

bool exist(char s, Set \* LA)  
{ bool b(false);  
 for(Set \* p = LA; p && !b; p = p->next ) b |= (s == p->el);  
 return b;  
}

Очевидно, что оба варианта реализации имеют линейную временную сложность по мощности множества: O(*nA*). Оценка не меняется и в том случае, если при обнаружении элемента в множестве цикл прерывается.

Для ввода и вывода множества символов удобным способом представления является строка — массив, ограниченный нулём. Такой массив можно ввести и вывести в один приём, без применения цикла и легко инициализировать константой:

cout ≪ "\n A="; cin ≫ A;  
cout ≪ " Введено A=" ≪ A ≪ '\n';

Наличие ограничителя позволяет отказаться от использования переменной для хранения мощности множества. Преобразование массива A в список LA выполняется по тривиальному алгоритму:

Set \*LA = nullptr;  
for( int i = 0; A[ i ]; ++i)   
 { Set \*temp = new Set;  
 temp->el = A[ i ]; temp->next = LA; LA = temp; }

Если немножко дополнить объявление структуры Set:

struct Set { char el; Set \* next;  
 Set(char e, Set \* n) : el(e), next(n) { }  
 ~Set( ) { delete next; }  
};

то преобразование массива в список станет ещё проще:

for( int i = 0; A[ i ]; ++i) LA = new Set(A[ i ], LA);

а для удаления списка из памяти не понадобится цикл, достаточно написать delete LA.

То, что при преобразовании массива в список порядок элементов меняется на противоположный, не имеет значения, поскольку речь идёт о множестве, где порядок элементов безразличен.

## 4.2. Представление множества отображением на универсум

Если элементы универсума упорядочить, т. е. представить в виде последовательности *U* = <*u*0*,u*1*, u*2*… um-*1>, то любое его подмножество *A* ⊆ *U* может быть задано вектором логических значений *C* = <*c*0*, c*1*, c*2 *… cm-*1>, где *ci*= {*ui ∈ A*}, или вектором битов *ci*= 1, если *ui ∈ A*, иначе — 0.

Такой способ представления множеств в памяти имеет практическое значение, если мощность универсума *m* = |*U*| не очень велика и существует простая функция *f* : *U* → [0 … *m* – 1] отображения элемента множества в соответствующий ему порядковый номер бита.

Так, например, если *U* — множество десятичных цифр, подходящей функцией будет *f*(*a*)= *a –* '0', где *a* — символьная переменная с кодом цифры, поскольку известно, что коды цифр образуют монотонную последовательность. Аналогично, для множества прописных латинских букв можно взять *f*(*a*) = *a – 'A'*. Для шестнадцатеричных цифр, коды которых образуют два интервала, функция будет сложнее: *f (a)* = *a* ≤'9' ? *a –* '0' *: a –* '*A*'+ 10. В общем случае можно использовать словарь — массив, содержащий универсум, например, для шестнадцатеричных цифр — U[ ] = "0123456789ABCDEF".

Операции над множествами в форме вектора битов сводятся к логическим операциям над соответствующими битами множеств. Для вычисления объединения *A* ∪ *B* следует выполнить *a* [ *i* ] || *b* [ *i* ], для пересечения *A* ∩ *B* — *a*[ *i* ] && *b*[ *i* ], для разности *A* \ *B — a*[ *i* ] && !*b* [ *i* ] для всех битов от *i*= 0 до *i*= *m*– 1. Следовательно, временная сложность двуместной операции с множествами *A* и *B* в форме вектора битов будет *O*(*m*), что при фиксированном *m* соответствует O(1), т. е. не зависит от мощности этих множеств.

Для получения вектора битов *bA* из строки символов *A* следует заполнить вектор *bA* нулями, а затем установить в 1 биты, соответствующие каждому символу из *A*:

for (int i = 0; A[ i ]; i++) bA [ f ( A[ i ] )] = 1.

Можно использовать автоматическое заполнение нулями, происходящее при объявлении массива с инициализацией:

int bA[m]{ };

Обратное преобразование очевидно:

for (int i = 0, k = 0; i < m; ++i) if ( bA[ i ] ) A[ k++ ] = f -1( i ),

где *f* -1(*i*) — функция, обратная для *f*(*a*). Так, если *f*(*a*) = *a* – '0', то *f* ‑1(*i*) = *i*+ '0'.

Можно, хотя и не обязательно, использовать для массива битов тип *bool*.

bool bA[m]{ };  
for (int i = 0; A[ i ]; ++i) bA [ f ( A[ i ] )] = true.

Использование массива битов в качестве промежуточной памяти при работе с множеством, представленным набором элементов — самый простой способ устранить дубликаты. Достаточно преобразовать массив (список) элементов в массив битов, а затем обратно. Затраты на такое преобразование будут иметь порядок *O*(n). Устранение же дубликатов «в лоб», т. е. проверка на уникальность каждого добавляемого в множество элемента, потребует *O*(*n*2) шагов.

Вектор битов может быть представлен в памяти в компактной форме — форме машинного слова, в качестве которого на языке С++ могут использоваться переменные целого типа *int, long, long long*. Для таких переменных в языке предусмотрены поразрядные логические операции:

— логическое сложение (поразрядное «ИЛИ») *A*| *B*, реализующее объединение множеств *A*∪*B*;

— логическое умножение (поразрядное «И») *A* & *B* — пересечение *A*∩ *B*;

— поразрядное сложение по модулю 2 (исключающее «ИЛИ», сравнение кодов) *A* ^ *B* — симметрическая разность *A* ⊕ *B* = (*A* ∪ *B*) \ (*A* ∩ *B*);

— инвертирование *~A*, соответствующее ­*Ā* — дополнению до универсума.

Операции над множествами в форме машинного слова выполняются за один шаг алгоритма независимо от мощности множеств, т. е. имеют временную сложность O(1). Например, вычисление *E* = (*A* ∪ *B* ∩ *C*) \ *D* реализуется оператором *wE* = (*wA* | *wB* & *wC*) & ~*wD*, где *wA*, *wB*, *wC*, *wD* — машинные слова, хранящие соответствующие множества.

Способ применим, если размер универсума *m* не превосходит разрядности переменной (8 для *char*, 16 для *short*, 16 или 32 для *int*, 32 для *long,* 64 для *long long*). Если *m* > 64, можно использовать несколько слов. Если *m* не равно размеру слова, часть битов слова не используется. Обычно это не вызывает проблем. Исключение: если переменная сравнивается с нулём для выявления пустого множества, нужно, чтобы неиспользуемые биты содержали 0.

Недостаток способа — в отсутствии удобного доступа к каждому биту машинного слова, как к элементу массива. Вместо этого приходится генерировать множество из одного элемента {*a*} сдвигом 1 на *f* (*a*) битов влево. Далее с помощью поразрядного «ИЛИ» можно добавить элемент в множество, а с помощью поразрядного «И» — проверить его наличие в нём. Так, для преобразования множества из строки символов в машинное слово можно использовать алгоритм

wA = 0;  
for ( int i = 0; A[ i ]; ++i ) wA = wA | (1 ≪ f(A));

*Примечание*. Если программа пишется для компилятора, поддерживающего разрядность данных *int* — 16, а мощность универсума больше (переменная *wA* имеет тип *long*), вместо константы 1 следует использовать 1*L*.

Для обратного преобразования (из машинного слова в строку символов) удобнее использовать сдвиг слова вправо и умножение на 1:

for ( int i = 0, k = 0; i < m; ++i) if ( (wA ≫ i) & 1) A[k++] = f -1(i).

Отметим, что элементы массива битов нумеруются справа налево, а биты машинного слова — слева направо. Поэтому, если для массива битов и для машинного слова используется одна и та же функция отображения *f*(*a*)*,* порядок битов в этих структурах данных будет противоположный.

Вместо операции сдвига иногда удобно применить словарь — массив из машинных слов с единицей в соответствующем разряде. Для универсума мощностью 10 массив будет выглядеть так:

W[ ] = { 1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80, 0x100, 0x200 }.

При желании константы в словаре можно расположить в противоположном или в произвольном порядке.

Машинное слово отличается от других способов хранения множеств ещё и тем, что в нём одновременно можно устанавливать или проверять несколько битов. Так, поразрядное «или» с константой 0x7FF превращает множество десятичных цифр в полное, а поразрядное «и» позволяет убедиться, что оно пустое, независимо от состояния неиспользуемых битов. Константа 0x155 позволит проделать это с подмножеством нечётных цифр {1, 3, 5, 7, 9}, а 0x2AA — чётных {0, 2, 4, 6, 8}.

## 4.3. Замечание о функциях — операциях над множествами

Двуместные операции над множествами можно (хотя и необязательно) реализовать в виде функций.

Так, функция для вычисления пересечения двух множеств, заданных массивами, ограниченными нулём, может выглядеть так:

void AND(char \*E, char A[ ], char B[ ]) //Вариант 1  
{ int k = 0; for(int i = 0; A[ i ]; ++i) if(exist(A[ i ], B) E[k++] = A[ i ]; }

или так:

char \* AND(char A[ ], char B[ ]) //Вариант 2  
{ int k = 0; char \* E = new char[N + 1];

for(int i = 0; A[ i ]; ++i) if(exist(A[ i ], B) E[k++] = A[ i ];

E[ k ] = 0; return E;

}

Сложность обоих вариантов — O(*n*2): функция *exist*( ) линейной сложности находится внутри цикла просмотра множества *A* из *n* элементов (*n* — средняя мощность множеств *A*, *B*, *E*). Какой вариант лучше? Я полагаю, первый. Множество *E* предполагается объявленным размером под универсум + 1 и инициализированным нулями, так что об ограничивающем нуле можно не беспокоиться.

Во втором варианте массив *E* создаётся в свободной памяти без возможности инициализации, поэтому приходится заботиться об ограничивающем нуле, и, что гораздо важнее, об освобождении этой памяти, когда работа с массивом *E* будет закончена. Это должна делать вызывающая программа.

Но последнее не всегда возможно. Так, при вычислении выражения *E* = *A* ∩ *B* ∩ *C* ∩ *D* «естественным» способом

E = AND(A, AND(B, AND(C, D)));

промежуточные результаты *C* ∩ *D* и *B* ∩ (*C* ∩ *D*) будут недоступны для удаления и останутся в памяти в качестве мусора.

*Примечание*. В последнем примере проблему утечки памяти можно решить с помощью следующего трюка:

E = AND(A, ABC=AND(B, CD=AND(C, D))); delete [ ]ABC; delete [ ]CD;

Здесь с помощью присваиваний промежуточным множествам даются имена, что делает возможным их удаление.

Точно такая же ситуация возникает при работе с массивами битов. Предпочтительная реализация выглядит так:

void AND(bool E[ ], bool A[ ], bool B[ ])  
{ for(int i = 0; i < Nmax; ++i) E[ i ] = A[ i ] && B[ i ]; }

Для варианта «машинные слова» функция тривиальна.

В варианте «списки» оба подхода оказываются функционально эквивалентны, потому что списки всегда создаются в свободной памяти:

void AND(Set \*&E, Set \*A, Set \*B) //Вариант 1  
{ for(Set \* p = A; p; p = p->next)   
 if(exist(p->el, B) E = new Set(p->el, E); }

или так:

Set \* AND(char A[ ], char B[ ]) //Вариант 2  
{ Set \* E = nullptr;  
 for(Set \* p = A; p; p = p->next)   
 if(exist(p->el, B) E = new Set(p->el, E);   
 return E;   
}

Здесь проблемным может оказаться первый вариант, потому что предполагается, что фактический указатель на результат *E* обнулён. Если это не так, то функция будет добавлять результат вычисления к имеющемуся содержимому списка *E*. Если же в намерения программиста это не входит, рекомендуется добавить в первый вариант функции *AND* первую строку:

delete E; E = nullptr;

Для передачи вызывающей программе нового значения указателя *E* соответствующий аргумент объявлен ссылкой.

*Контрольные вопросы*

1. Какой способ размещения множества в памяти — самый экономный?
2. Какой способ размещения множества в памяти — самый универсальный?
3. Какой способ размещения множества в памяти — самый удобный для проверки принадлежности элемента множеству?
4. Какой способ размещения множеств в памяти обеспечивает самый быстрый алгоритм выполнения двуместной операции над ними?
5. При какой форме представления множеств в памяти можно задать множество-константу?
6. Какой способ представления в памяти множества символов самый удобный для ввода с клавиатуры?
7. Сколько различных подмножеств можно получить для множества мощностью *n*?

# 5. Генерация тестов

Программу, проверенную на тестах, введённых вручную или из специально подготовленного файла, можно затем дополнительно проверить подачей на вход некоторого количества случайных тестов, генерацию которых разумно поручить машине.

### *5.1. Генерация случайного подмножества*

Достаточно просто получить случайное множество в форме машинного слова. Для этого можно использовать функцию *rand*( ) из стандартной библиотеки *stdlib*. Функция возвращает псевдослучайное целое в интервале 0…*MAXINT*. Случайное слово из *m* битов (*m* — мощность универсума) можно получить, выделив его из возвращаемого значения. Так, для *m* = 10 это будет:

w = rand( ) %0x3FF.

А можно просто положить *w* = *rand*( ) и игнорировать лишние биты.

Если разрядность *int* равна 16, для получения слова типа *long* можно использовать функцию дважды и объединить возвращаемые значения:

w = static\_cast<long> ((rand( ) ≪ 16) | rand( )).

Если мощность универсума превышает разрядность машинного слова, генерируется несколько слов.

Можно вместо нескольких слов сгенерировать массив из *n* случайных битов:

for ( int i = 0; i < m; ++i) X[ i ] = rand( ) % 2.

*Замечание*. Последний способ в общем случае следует применять с осторожностью, так как младший бит генерируемого машинного слова при плохом генераторе может оказаться не вполне случайным, например, чередованием 0 и 1.

Следует отметить, что датчик *rand* ( ) даёт не случайные, а псевдослучайные числа. При каждом новом запуске программы последовательность этих чисел будет повторяться. Это очень удобно для отладки. В начало функции *main*( ) нужно вставить строку *srand*(*start*), которая обеспечит запуск датчика с указанного значения *start* (целого числа). Подбором значения start можно добиться удовлетворительного теста, необходимого для полноценной отладки. Когда отладка закончена, константу *start* заменяют вызовом функции из стандартной библиотеки *time.h*, возвращающей текущее время: *time*(*nullptr*).

Случайное множество в форме массива или списка проще всего получается генерированием последовательности битов:

int k = 0; Set \*LA = nullptr;  
for ( int i = 0; i < m; i++) if (rand( )%2)  
 { S[ k++ ] = f-1( i ); Set \*LA = new Set( f-1( i ), LA); }  
S[ k ] = 0;

Здесь, как и ранее, *f*-1( *i* ) — функция, преобразующая позицию *i* массива битов в элемент универсума.

Результат — строка символов *S* и список *LA* — множество со случайной мощностью *k* ∈ [0…*m* – 1].

Все рассмотренные генераторы создают множества, в которых каждый элемент универсума появляется с вероятностью 0,5. Может получиться и пустое, и полное множество, но в среднем мощность получается близкой к *m* / 2. Если требуется получать множества почти пустые или почти полные, нужно сделать так, чтобы вероятности появления 0 или 1 различались. Например, генератор массива битов (булевских значений) может выглядеть так:

for ( int i = 0; i < m; ++i) X[ i ] = static\_cast<bool>(rand( ) % p > q).

В этом генераторе вероятность появления 1 зависит от соотношения значений констант *p* и *q*. Так, например, при *p* = 5 датчик будет давать с равной вероятностью элементы множества {0, 1, 2, 3, 4}; если выбрать *q* = 3, вероятность генерации *true* будет 0,2, а при *q* = 0 — 0,8.

### *5.2. Генерация последовательности всех подмножеств заданного множества*

Подача на вход алгоритма последовательности всех подмножеств некоторого множества *X* может потребоваться для полного тестирования алгоритма или для решения задачи полным перебором в случаях, когда эффективного алгоритма не существует. Если мощность множества |*X*| = *m*, мощность множества всех его подмножеств — булеана (общее количество тестов) |2*X*| = 2*m*.

Если *m* ≤ 32, последовательность подмножеств проще всего получить в форме машинных слов по очевидному алгоритму:

for ( w = 0; w < 2m; ++w) **yield**(w).

Здесь и далее ***yield***(*w*) — некоторая функция, использующая множество *w*.

Для практических целей часто бывает удобнее, чтобы каждое подмножество в последовательности отличалось от предыдущего появлением или исчезновением ровно одного элемента (*m*-битный код Грея).

а) вариант для массива битов

void GGen (int m, int A[ ])   
{  
 for (int i = 0; i < m; ++i) A[ i ] = 0; // Исходный код (из нулей)  
 int i = 0; // Счётчик кодов  
 do {  
 **yield**(A); //Использовать очередной код  
 ++i; int p( 0 ), j( i );  
 while ( (j & 1) == 0) { j = j/2; ++p; } // Искать младший бит != 0  
 if ( p < n ) A[ p ] = !A[ p ]; // и инвертировать его  
 } while ( p < m);  
}

б) вариант для машинного слова

for ( int i = 0; i < 2m; i++) { w = i ^ (i ≫ 1); **yield**(w); }.

### *5.3. Множества с повторениями. Тест — аналог кода Грея*

Для некоторых задач необходимо обеспечить возможность нескольких вхождений элемента множества в тестовый набор. В этом случае можно считать, что источником тестовых наборов является *множество с повторениями*, или *мультимножество*, в котором каждый элемент может появляться несколько раз. Число возможных вхождений является существенным и носит название *кратности* элемента в множестве. Множество с повторениями, содержащее, например, элемент *a* кратности 2, элемент *b* кратности 3 и элемент *c* кратности 1, можно обозначить как { *a*, *a*, *b*, *b*, *b*, *c* } или { 2\**a*, 3\**b*, 1\**c* }.

Порядок элементов не существенен, существенна только кратность: { *a*, *a*, *b*, *b*, *b*, *c* } = { *a*, *b*, *a*, *b*, *c*, *b* } ≠ { *a*, *b*, *c* }.

Для обычных множеств { *a*, *a*, *b*, *b*, *b*, *c* } = { *a*, *b*, *c* }.

Мощность множества с повторениями равна сумме кратностей его элементов.

Для представления множества с повторениями в памяти, кроме набора элементов, можно использовать массив кратностей, являющийся обобщением массива битов. Универсум M такого множества — это массив максимальных кратностей. Располагая таким массивом, можно построить алгоритм генерации последовательности множеств с повторениями, отличающимися друг от друга добавлением или удалением одного элемента — аналог кода Грея для мультимножеств.

void MGen(int m, int M[ ]) // Все подмножества мультимножества   
{ // M в виде последовательности  
 // с минимальным кодовым расстоянием  
 int \*D = new int[ m ], \*S = new int[ m ];  
 for (int j = 0; j < m; ++j) { S[ j ] = 0; D[ j ] = 1; } //инициализация  
 do {  
 **yield**(S); //Использовать *S*  
 int j = 1, i = 0;  
 while ( j ) {  
 if (D[ i ])  
 { if (S[ i ] == M[ i ]) { D[ i ] = 0; --i; } // счёт назад  
 else j=0;  
 }  
 else {  
 if (S[ i ] == 0) { D[ i ] = 1; ++i; } // счёт вперёд  
 else j = 0;  
 }  
 }  
 if (i<n) if (D[ i ]) ++S[ i ]; else --S[ i ]; //счёт  
} while (i < n);  
delete [ ] D; delete [ ] S;  
}

### *5.4. Случайное подмножество заданной мощности*

Все рассмотренные ранее датчики генерировали множество случайной мощности. Если же требуется случайное подмножество заданной мощности *k*, например, в форме массива, его иногда пытаются получить следующим алгоритмом:

for ( int i = 0; i < k; ++i ) X[ i ] = rand( ) % m.

Этот способ не годится, потому что он даёт не множество, а последовательность, в которой возможны повторы, и их будет много, если *k* близко к *m*, т. е. фактическая мощность множества будет меньше заданного *k*.

Можно усовершенствовать этот алгоритм: повторять генерацию очередного элемента множества до тех пор, пока не кончатся совпадения с уже имеющимися. Способ рекомендуется при больших *m* (*k* ≪ *m*). Если же *m* не намного больше *k*, то с ростом *k* вероятность получить новый элемент множества очень быстро уменьшается, а при *k* = *m* алгоритм может вообще никогда не остановиться.

Способ, рекомендуемый для небольших *m*: сформировать в памяти для результата массив — универсум, на каждом шаге убирать сгенерированный элемент множества в его начало и разыгрывать оставшиеся. Результат будет получен за время *O*(*k*).

for ( int i = 0; i < m; ++i ) X[ i ] = i + 1;   
 //Формирование универсума {1 … *m*}  
for ( int i = 0; i < k; ++i) // Генерация подмножества мощностью *k*  
{ int p = rand( ) % (m – i); // Случайный выбор среди оставшихся  
 if (p) Swap ( X[ i + p ], X [ p ] ); // Если *p* ≠ 0, обменять.  
 }.

Результат — первые *k* элементов массива *X*. Способ легко приспособить для генерации последовательности подмножеств нарастающей мощности: использовать массив *X* в качестве теста после каждого добавления в него очередного элемента.

Если взять *k* = *m* – 1, получается *алгоритм генерации случайной перестановки*. Без ограничения общности он может использовать очередную перестановку для генерации следующей, не требуя для новой перестановки обязательного перезапуска датчика случайных чисел.

### *5.5. Последовательность всех подмножеств заданной мощности*

Подавать на вход алгоритма последовательность всех подмножеств некоторого множества можно разными способами.

Иногда может быть удобен лексикографический порядок последовательности подмножеств. Если разметить универсум и работать с номерами, то лексикографический порядок — это последовательность, упорядоченная по возрастанию образуемых этими номерами чисел. Так, для *m* = 8 и *k* = 4 это будет:

1 2 3 4 1 2 3 5 1 2 3 6 1 2 3 7 1 2 3 8 1 2 4 5 1 2 4 6  
1 2 4 7 1 2 4 8 1 2 5 6 … 5 6 7 8.

**K-элементные подмножества — лексикографический порядок**

void GenK (int n, int k, int S[ ])   
 // K-элементные подмножества  
{ int i, p; // в лексикографическом порядке  
 for (i=0; i < k; ++i) S[ i ] = i + 1;  
 do {  
 **yield**(S); // Использовать *S*   
 if (S[ k-1 ] == n) --p;  
 else p = k - 1;  
 if (p >= 0)  
 for (i = k - 1; i >= p; --i)  
 S[ i ] = S[ p ] + i - p + 1;  
 } while (p >= 0);  
}

Для генерации последовательности подмножеств с минимальным кодовым расстоянием, т. е. отличающихся на каждом шаге заменой ровно одного элемента, необходим более сложный алгоритм. Он получает полный набор таких последовательностей и выводит каждую на экран. Алгоритм сводит задачу для последовательности из *m* по *k* к комбинации последовательностей меньшей мощности, вычисляемых рекурсивно. Результат выводится на экран.

**K-элементные подмножества — последовательность с минимальным кодовым расстоянием**

void GenPk (int n, int k, int S[ ], int &p) // K-элементные подмножества  
{ int \*B, i, j, p1, p2; // в виде последовательности  
 // с минимальным кодовым расстоянием  
 if ((k == 1) || (k == n))  
 { for (i = 0; i < n; ++i) S[ i ] = i + 1; // Тривиальный случай  
 if (k == 1)  
 { p = n; for(i = 0; i < n; ++i) cout << A[i];}  
 else {p = 1; for(i = 0; i < n; ++i) cout << A[i];}  
 cout << "!";  
 }  
 else {  
 GenPk (n-1, k-1, A, p2); //Получить и запомнить окончание  
 B = new int[ p2 \* (k - 1) ];   
 for (i = 0; i < p2 \* (k - 1); ++i) B[ i ] = S[ i ];  
 GenPk(n-1, k, A, p1); // Получить начало  
 for( i = 0; i < p2; ++i) // и присоединить окончание  
 { for ( j = 0; j < k - 1; ++j)  
 S[ p1 \* k + i \* k + j ] = B[ (p2 - (i + 1)) \* (k - 1) + j ];  
 S[ p1 \* k + i \* k + k – 1 ] = n;  
 }  
 delete [ ] B;   
 p = p1 + p2; //Мощность результата  
 for ( i = 0; i < p; ++i) { //Результат  
 for (j = 0; j < k; ++j) cout << A[ i \* k + j ];  
 cout << "; ";  
 }  
 }  
 cout << endl;  
}

### *5.6. Генерация перестановок*

Некоторые алгоритмы требуют подачи на вход полного множества *X* в виде последовательности, отличающейся порядком расположения элементов. Пример такого алгоритма — проверка двух графов одинаковой мощности на изоморфизм, заключающаяся в подборе такой нумерации вершин второго графа, чтобы его рёбра совпали с рёбрами первого графа. Функция *Neith*( ) генерирует все перестановки множества чисел от 1 до m в виде последовательности, в которой на каждом шаге меняются местами два смежных элемента.

inline void Swap( int &p, int &q) { int r (p); p = q; q = r; }  
void Neith( int n )   
{ int \*X = new int[ m ], \*C = new int[ m ], \*D = new int[ m ], i, j, k, x;  
 for (i = 0; i < m; ++i) // Инициализация  
 { X[ i ] = i + 1; C[ i ] = 0; D[ i ] = 1; }  
**yield** (X); // Использование первой перестановки  
C[ m – 1 ] = –1; i = 0;  
while ( i < m – 1 ) // Цикл перестановок  
{ i = 0; x = 0;  
 while (C[ i ] == (m – i – 1))  
 { D[ i ] = !D[ i ]; C[ i ] = 0; if (D[ i ]) ++x; ++i; }  
 if (i < m – 1) // Вычисление позиции k и перестановка смежных  
 { k = D[ i ] ? C[ i ] + m : m – i – C[ i ] + x – 2;  
 Swap( X[ k ], X[k + 1] ); }  
 yield (X); // Использование очередной перестановки  
 ++C[ i ];  
 }  
}.

*Контрольные вопросы*.

1. Какой способ размещения множеств в памяти является оптимальным для генерации кода Грея?
2. Получена последовательность кодов Грея: 0000 0001 0011 0010  
   Укажите следующий по порядку код.
3. Какова временная сложность оптимального алгоритма генерации произвольного случайного подмножества?
4. Какова временная сложность оптимального алгоритма генерации случайного подмножества заданной мощности?
5. Какова временная сложность оптимального алгоритма генерации всех перестановок элементов множества?

# 6. Множество — пользовательский тип данных

Если некоторая структура данных, например, массив, используется как реализация множества, это означает, что программист просто устанавливает для себя некоторые правила для работы с этим массивом и последовательно их придерживается. Часто большего и не требуется. Однако можно рассматривать множество как абстрактную структуру данных — область памяти, доступ к которой возможен только через некоторый интерфейс, т. е. набор функций, специально созданных для работы с этой памятью. Язык С++ поддерживает работу с абстрактными данными через механизм классов: абстрактная структура данных определяются как класс, в котором задаются как данные, так и связанные с ними операции. Определение класса позволяет расширить язык C++, включив в него множество как пользовательский тип данных.

Рассмотрим пример — класс для работы с множеством, представленным массивом символов (строкой). Кроме массива символов, в котором будет храниться множество, в класс будет естественно включить переменную *n* — мощность множества и *S* — символ-тег, с помощью которого множество можно идентифицировать.

class Set {  
private:  
 const int Nmax;  
 int n; char S;  
 char A[Nmax+1];  
public:  
 void AND(const Set &A, const Set &B); // (1)  
 Set AND(const Set &B) const; // (2)  
 friend void AND(Set &E, const Set &A, const Set &B); // (3)  
 bool exist(char s) const { bool b(false);  
 for(int i = 0; A[ i ] && !b; ++i) b |= A[ i ] == s;  
 return b; }   
 int power( ) const { return n; }  
//…  
} X1, X2, X3;

Класс *Set* является пользовательской структурой данных. Объявляя класс, можно (но не обязательно) объявить и объекты соответствующего типа: *X1*, *X2*, *X3*.

Синтаксис объявления класса — такой же, как и для структуры struct. Более того, ключевые слова *class* и *struct* в этом контексте полностью взаимозаменяемы. Отличие только в том, что всё содержимое класса *class* по умолчанию закрыто (*private*), а структуры *struct* — открыто (*public*). Тем самым обеспечивается обратная совместимость с программами на Си. На практике ключевое слово *struct* применяется для объявления простейших классов, в которых не предполагается ничего закрывать.

Назначение закрытой части — запретить произвольное использование объявленных в ней идентификаторов. Защита обеспечивается компилятором. Так, при попытке вычислить среднюю мощность объявленных множеств: (*X1*.*n* + *X2*.*n* + *X3*.*n*)/3 компилятор укажет на ошибку и сообщит «недоступно».

Для доступа к закрытым полям нужно предусмотреть соответствующие функции-члены. Такую роль выполняет функция power(). Следующее утверждение будет компилироваться:

int middle\_power = (X1.power( ) + X2.power( ) + X3.power( ))/3;

Функция *power*( ) позволяет узнать значение переменной n, но не позволяет изменять его. Если нужна возможность изменения, функция должна возвращать ссылку:

int & power( ) { return n; }

Но это считается плохой практикой. Возможность изменения проще обеспечить размещением переменной в открытой части класса.

В объявлении класса *Set* в качестве примера объявлены три варианта функции, вычисляющей пересечение двух множеств. Определить эти функции можно так:

void Set ∷ AND(const Set & A, const Set & B) // (1)  
{ int k = 0;   
 for(int i = 0; A.A[ i ]; ++i) if(B.exist(A.A[ i ]) A[k++] = A.A[ i ]; }   
Set Set ∷ AND(const Set & B) // (2)  
{ Set E;  
 for(int i = 0; A[ i ]; ++i) if(B.exist(A[ i ]) E.A[E.n++] = A[ i ];  
 E.A[ n ] = 0; return E;   
}   
void AND(Set & E, const Set & A, const Set B) // (3)  
{ E.n = 0; for(int i = 0; A.A[ i ]; ++i)   
 if(B.exist(A.A[ i ], B) E.A[E.n++] = A.A[ i ];  
 E.A[E.n] = 0; }

Если сравнить два варианта функции того же назначения, обсуждённые ранее, можно видеть, что изменилось в объявлении и определении этих функций при работе с классом *Set*.

1. У функций в вариантах (1) и (2) исчез первый аргумент. Теперь его роль исполняет объект, для которого функция вызывается. Вызов для вычисления *X1* = *X2* ⋂ *X3* может выглядеть так:

X1.AND(X2, X3); // (1)  
X1 = X2.AND(X3); // (2)  
AND(X1, X2, X3); // (3)

1. Аргументами функций теперь являются объекты. Объявление таких аргументов обычным образом (просто заменой *char* [ ] на *Set*) приведёт к тому, фактические аргументы будут копироваться в формальные. Это возможно, но нежелательно. Для исключения копирования аргументы передаются как ссылки, а для того, чтобы защитить их от изменения в функции, применяются модификаторы *const*. Модификатор *const* в заголовке функции после скобок означает, что функция не меняет и объект, для которого вызывается. ВАЖНО: фактическим параметром функции, принимающей константную ссылку, может быть временный объект, образующийся в результате вычислений. Например, возможен следующий вызов:

X1.AND( X2.AND( X3.AND( X4 ) );

1. Функция-член имеет доступ ко всем полям своего объекта (*n*, *S*, *A*), как глобальным переменным, без всяких дополнительных указаний. Если объект — единственный аргумент, другие аргументы не нужны.
2. Функция (3) с модификатором *friend* — это функция-друг. Она вызывается как обычная функция, без неявного аргумента-объекта, но хотя бы один её аргумент должен быть объектом соответствующего типа. Функция-друг имеет те же права на доступ к внутренней части класса, что и функция-член.

Все три варианта функции *AND* могут быть определены одновременно (т. е. перегружены) и затем использоваться по усмотрению программиста.

Определить функцию можно как внутри класса (функции *exist*, *power*), так и вне его. ВАЖНО: функции, определённые внутри объявления класса, по умолчанию считаются встроенными, или макросами (*inline*). Так, вместо вызова функции *power*( ) будет просто подставлено поле *n* соответствующего объекта.

Хорошая практика: определять простые функции внутри определения класса, а более сложные — отложить, чтобы определение класса не было слишком громоздким и легко читалось.

Модификатор *inline* при желании можно указывать явно. Встраивание функции в любом случае не гарантируется. Более того, современные компиляторы часто игнорируют указание *inline* и могут самостоятельно сделать одни вызовы функции встроенными, а другие — нет, в зависимости от контекста вызова.

# 7. Типы классов и служебные функции-члены

Каждое объявление класса означает и объявление служебных функций-членов, предоставляемых системой по умолчанию. Для каждой из этих функций программист может, а часто и обязан предоставлять свою версию.

В первую очередь это функция-конструктор, автоматически вызываемая при создании объекта. Имя этой функции совпадает с именем класса, тип возвращаемого значения никогда не указывается. Назначение конструктора — инициализировать все поля объекта как подготовка к их использованию. Так, конструктор для приведённого выше класса Set может выглядеть так:

Set( ) : Nmax(26), n(0), S('R') { A[0] = 0; }

Важная особенность конструктора, отличающая его от других функций — наличие **списка инициализации**, состоящего из последовательности полей и присваиваемых им значений. Присваивание значений возможно и в теле конструктора, но этот способ рекомендуется применять, только если непосредственная инициализация невозможна или неудобна. ВАЖНО: порядок инициализации полей всегда совпадает с порядком их объявления, поэтому менять их порядок в списке инициализации не рекомендуется. Если в классе есть поля с модификатором const, список инициализации — единственный способ присвоить им значения, поскольку присваивание для таких полей запрещено.

В классе может быть определено любое количество конструкторов, различающихся типом и/или количеством аргументов. Например, кроме приведённого выше конструктора пустого множества, можно иметь конструктор множества случайной мощности:

Set(char s ) : Nmax(26), n(0), S(s) {   
 for(auto i = 0; i < Nmax; ++i) if(rand( )%2) A[n++] = U[ i ];   
}

Этот конструктор отличается от предыдущего наличием аргумента — тега создаваемого множества. Он использует датчик случайных чисел, делая *Nmax* попыток. Если выпадает 1, к множеству добавляется *i*-ый символ из словаря *U*[ ], содержащего универсум.

Конструктор без аргументов — это важный частный случай, называемый конструктором **по умолчанию**. Конструктор, предоставляемый системой по умолчанию, ничего с объектом не делает, поэтому его обязательно нужно определять. Иногда это бывает невозможно, поэтому действует правило: любой конструктор с аргументом делает конструктор по умолчанию недоступным. С другой стороны, наличие конструктора по умолчанию — обязательное условие для создания из объектов этого класса массивов. Проблема может быть снята явным объявлением конструктора по умолчанию:

Set( ) = default;

Этот конструктор сам по себе ничего не делает, но он запускает конструкторы для отдельных полей, если они являются объектами некоторого класса.

Современные компиляторы (С++17 и выше) разрешают инициализацию полей непосредственно в объявлении класса:

class Set {  
private:  
 const int Nmax = 26;  
 int n = 0; char S = 'R';  
 char A[Nmax+1];  
public: …

Это означает, что соответствующие инициализаторы будут включены во все конструкторы, с правом программиста при необходимости изменить инициализацию по умолчанию.

Проблему конструктора по умолчанию можно решить также указанием значений аргументов по умолчанию, например, так:

Set(int n = 0; char s = 'R' ) : Nmax(26), n(n), S(s) { }

Такой конструктор может быть вызван с двумя аргументами Set(10, 'A'), с одним аргументом Set(10) или вообще без аргументов, как конструктор по умолчанию. Заметим, что это не три разных функции, а одна функция с тремя способами вызова.

Кроме конструктора по умолчанию, система предоставляет каждому классу конструктор копии, перегрузку присваивания и деструктор.

Конструктор копии используется для создания нового объекта копированием существующего. Это конструктор с аргументом — константной ссылкой на объект того же типа.

Set(const Set &);

Такой же аргумент имеет и перегрузка присваивания:

Set & operator = (const Set &);

Перегрузка присваивания отличается тем, что вызывается для уже существующего объекта. Обе функции по умолчанию копируют свой аргумент в создаваемый или существующий объект по принципу байт в байт.

Деструктор — это функция с именем, совпадающим с именем класса, которому предшествует знак '~':

~Set( );

Для этой функции также никогда не указывается тип возвращаемого значения. Список аргументов у неё всегда пуст, поэтому функция существует в единственном варианте, и перекрытие её невозможно. Назначение функции — действия при уничтожении объекта. Если такие действия нужны, функцию следует определить, потому что по умолчанию она ничего не делает.

Существуют **три типа классов**, различающихся необходимым набором служебных функций:

— класс-понятие;

— класс-значение;

— класс-контроллер.

**Класс-понятие** — это класс без данных (обычно *struct*, т. к. в закрытую часть такого класса нечего помещать). Примеры:

struct input\_iterator\_tag{ };  
struct Error{ };  
struct less { bool operator( )(int a, int b){ return a < b; }

С такими классами мы встретимся во второй части курса. Первый пример — это тег итератора, с помощью которого задаются его свойства. Второй пример — это объявление класса ошибки. Такой класс используется как аргумент оператора возбуждения исключения «throw Error» и его перехвата «catch (Error)». Третий пример — это объявление функтора, использующегося как аргумент для алгоритмов, в которые нужно передавать функцию.

Для класса-понятия рассмотренные выше служебные функции вообще не требуются.

**Класс-значение** (простейший класс в терминологии разработчика языка С++ Б. Страуструпа) — это класс, имеющий поля-данные.

Такому классу необходимы только конструкторы. Конструктор копии и перегрузку присваивания для них тоже можно определить, но делать этого не следует без особых причин, потому что функции, предлагаемые по умолчанию, всегда работают быстрее. Деструктор для такого класса не нужен, т. к. ему нечего делать.

**Класс-контроллер** — это класс, управляющий ресурсом. Другое название такого класса — дескриптор ресурса. Именно такие классы обсуждаются в большинстве книг по программированию, оставляя первые два типа в тени.

**Ресурс** — это нечто вне класса. Чаще всего это память, но может быть и файл, канал связи и др.

ВАЖНО: для обеспечения устойчивости программы к сбоям ресурс, управляемый классом, должен быть **единственным**. И наоборот, если программе требуется ресурс, следует объявить класс, который будет им управлять. Смысл этих правил будет подробно рассмотрен в теме «Исключения» во второй части курса.

Рассмотренный выше пример класса для множества в массиве — это класс-значение. Его можно переделать в класс-контроллер, если перенести множество-массив в свободную память.

class Set {  
private:  
 const int Nmax;  
 int n; char S;  
 char \*A;  
public:  
Set( ) : Nmax(26), n(0), S('R'), A(new char[Nmax]){ }  
~Set( ) { delete [ ] A; }  
…

В этом классе вместо массива хранится указатель на него. Конструктор инициализирует указатель адресом в свободной памяти, где располагается массив, а деструктор будет освобождать эту память при уничтожении объекта.

Для класса-контроллера определение конструктора копии и перегрузки присваивания является **обязательным**, потому что ресурс по умолчанию не копируется, и это приводит к тому, что в результате копирования или присваивания появляются два объекта, управляющие одним и тем же ресурсом. Этого можно не заметить, если для класса не определён деструктор. Тогда при уничтожении исходного объекта ситуация исправляется, и программа нормально работает. Но при окончательном удалении объектов их ресурсы остаются в памяти в качестве мусора. Если деструктор определён, тогда удаление исходного объекта удаляет и ресурс, и указатель в объекте копии становится недействителен. Удаление объекта-копии сопровождается попыткой повторного удаления уже не су­ще­ству­ющего ресурса, что приводит к аварийному завершению программы.

Для рассмотренного выше примера конструктор копии и перегрузка присваивания могут выглядеть так:

Set(const Set & other ) : Set( ) {  
 n = other.n; S = other.S; strcpy(A, other.A); }  
Set & operator = (const Set & other) {  
 if (this != &other) {  
 n = other.n; S = other.S; strcpy(A, other.A);  
 }  
 return \*this;  
}

Конструктор копии должен делать то же самое, что и конструктор по умолчанию, т. е. создавать в памяти массив для множества. Это можно обеспечить вызовом последнего в списке инициализации, чтобы заставить его выполниться перед собственно копированием. Такой приём называется *делегирование конструкторов*. Он доступен для версий С++17 и выше.

Перегрузка присваивания имеет дело с уже существующим объектом. Она начинается с проверки на **самоприсваивание**. Здесь используется указатель *this* = адрес объекта слева от знака присваивания. Если этот адрес не совпадает с адресом аргумента, производится копирование содержимого всех полей объекта. В обоих случаях предполагается, что множество представлено строкой в стиле Си, т. е. массивом символов, ограниченном нулём.

Создание копии ресурса при копировании объекта иногда может быть избыточным. Если при передаче аргументов в функцию проблема снимается заменой значения константной ссылкой, то при возврате результата из функции дублирование ресурса особенно неприятно, потому что создаваемый при этом временный объект в итоге уничтожается вместе с вновь созданным ресурсом.

Для снятия этой проблемы в версию С++11 были добавлены **перемещающий** конструктор и **перемещающее** присваивание, которые не создают копию ресурса, а просто передают его в результат. Для обеспечения такой возможности в язык была добавлена операция '&&' — правая ссылка (*r-value*), а существующий знак операции '&' стал обозначать левую, или универсальную ссылку (*l-value*). Перемещающие варианты копирования и присваивания для рассматриваемого примера класса могут выглядеть так:

Set(Set && other ) : n(other.n), s(other.S), A(other.A) {  
 other.A = nullptr; }  
Set & operator = (Set && other) {  
 if (this != &other) {  
 n = other.n; S = other.S; A = other.A; other.A = nullptr;  
 }  
 return \*this;  
}

Перемещающий вариант копирования или присваивания выбирается компилятором в случае, когда копируется временный объект. Такой выбор можно форсировать, если применить служебную функцию *move*:

Set A(std∷move(B));

при создании нового объекта из существующего и

return std∷move(C);

при возврате значения функции. Здесь *std*∷*move*( ) — это не функция в обычном понимании, а средство преобразования универсальной ссылки в правую.

Обсудим теперь вопрос, класс какого типа следует использовать в эксперименте с множествами в памяти ЭВМ.

Очевидно, что множество в машинном слове может быть реализовано только как класс-значение, а множество в списке — только как класс-контроллер. Для массива и массива битов возможны оба варианта. Есть основание предполагать, что вариант с классом-контроллером будет работать быстрее за счёт исключения ненужного копирования массивов.

Каждое объявление класса означает и объявление служебных функций-членов, предоставляемых системой по умолчанию. Для каждой из этих функций программист может, а часто и обязан предоставлять свою версию.

Возможный вариант к**ласса-множества на основе массива** с полным набором функций-членов

class Set {  
private: // Закрытая часть класса — данные  
static int N; // мощность универсума  
 int n; // мощность множества  
 char S, \*A; // тег и память для множества  
public: // Открытая часть — функции для работы с множеством  
 Set operator | (const Set&) const; // объединение  
 Set operator & (const Set&) const; // пересечение  
 Set & operator |= (const Set&); // объединение и присваивание  
 Set & operator &= (const Set&); // пересечение и присваивание  
 Set operator ~ ( ) const; // дополнение до универсума  
 void Show( ); // вывод множества на экран  
 operator int ( ) { return n; } // получение мощности   
 explicit Set(char); // конструктор множества  
 Set( ); // ещё конструктор — по умолчанию  
 Set(const Set &); // конструктор копии  
 Set(Set &&); // конструктор копии с переносом (C++11)  
 Set operator = (const Set &); // оператор присваивания  
 Set operator = (Set &&); //оператор присваивания с переносом (C++11)  
 ~Set( ) { delete [ ] A; } // деструктор  
};

Имя класса *Set* — это имя нового типа данных. С его помощью будем объявлять в программе множества-объекты.

Память для множества находится в закрытой части класса и доступна через член *A* — указатель на символы. Размер памяти не определён. Кроме этого, в закрытую часть помещены вспомогательные переменные-члены: мощность универсума *N*, текущая мощность множества *n* и символ-тег *S*, с помощью которого можно различать объекты-множества. Мощность универсума *N* объявлена со спецификатором «*static*». Это означает, что все объекты класса *Set* будут использовать единственную копию этой переменной. Переменная *N* должна быть дополнительно объявлена вне всех функций, чтобы ей была выделена память. При этом требуется установить и её значение:

int Set ∷ N = 26; // Мощность универсума для множества латинских букв.

В открытой части класса объявлены функции-члены, с помощью которых в программе-клиенте можно работать с множеством. Каждая функция-член имеет в качестве обязательного аргумента объект, для которого она вызывается. Данные-члены из закрытой части класса доступны в ней как обычные глобальные переменные, и их тоже не нужно передавать как аргументы. Всё это позволяет свести количество аргументов функций-членов к минимуму или даже совсем от них отказаться, не засоряя при этом пространство глобальных имён.

Для работы с множествами-массивами предполагается использовать такой же синтаксис, как для машинных слов. С этой целью функции объединения, пересечения и дополнения множеств объявлены с именами, содержащими ключевое слово «*operator*», после которого следует знак соответствующей операции. Операции языка С++ «|», «&» и «~» определены так, чтобы их можно было использовать в выражениях, состоящих из данных типа *Set*. Такой приём называется перегрузкой операций. Чтобы это действительно можно было возможно, функции объявлены так, чтобы была обеспечена совместимость со встроенными операциями языка С++: все функции возвращают объект типа *Set*, а двуместные операции в качестве аргумента (второго, потому что первый — это сам объект) имеют константную ссылку на объект типа *Set*. Функции не меняют объект, для которого вызываются. Для контроля за этим в каждом из объявлений после списка параметров помещён спецификатор *const*.

Если мы объявляем для своего типа данных для операции пересечения перегрузку знака «&», а также перегрузку присваивания «=», то это не делает автоматически доступной комбинированную операцию «&=» — пересечение и присваивание. Такую операцию нужно тоже перегружать явно. Более того, рекомендуется обязательно сделать это и сделать согласованно. Проще всего этого добиться, реализую двуместную операцию через комбинированную:

Set& Set :: operator &= (const Set & B)  
{ Set C(\*this);  
 n = 0;  
 for (int i = 0; i < C.n; ++i) {  
 for (int j = 0; j < B.n; j++)  
 if (C.A[ i ] == B.A[ j ]) A[ n++ ] = C.A[ i ];  
 }

A[ n ] = 0; // ограничитель строки  
 return \*this;   
}

Set Set :: operator & (const Set & B) const  
{ Set C(\*this);  
 return (C &= B);   
}

В первой функции объявляется множество *C*, которое конструктор копии заменяет текущим множеством, для которого вызвана операция (множеством слева от присваивания). Затем текущее множество делается пустым, и в нём формируется результат пересечения временного объекта *C* и множества *B*. Поскольку для этого используется двойной цикл по мощности множеств, временная сложность операции — квадратичная. Результат — текущий объект. Во избежание лишнего копирования можно в качестве результата вернуть ссылку на него.

Вторая функция — двуместная операция пересечения множеств — тоже сперва создаёт копию текущего объекта, а затем возвращает результат комбинированной операции с временным объектом в левой части. Возврат ссылки здесь недопустим: по выходе из функции временный объект автоматически уничтожится, и возвращённая ссылка станет недействительна.

Операция объединения множеств «|» реализуется похожим алгоритмом:

Set & Set :: operator |= (const Set & B)  
{ for(int i = 0; i < B.n; ++i) {  
 bool f = true;  
 for (int j = 0; j < n; ++j)  
 if (B.A[ i ] == A[ j ]) f = false;  
 if (f) A[ n++ ] = B.A[ i ];  
 }  
 A[ n ] = 0;  
 return \*this;  
}  
Set Set :: operator | (const Set & B) const  
{ Set C(\*this);  
 return (C |= B);   
}

В текущий объект-множество добавляются недостающие элементы из *B*.

Операция вычисления дополнения может быть реализована так:

Set Set :: operator ~ ( ) const  
{ Set C;  
for (char c = 'A'; c <= 'Z'; ++c) { //Вариант для латинских букв‼!  
 **bool f = true;** for (int j = 0; j < n; ++j)  
 if (c == A[ j ]) { f = false; break; }   
 if (f) C.A[ C.n++ ] = c;  
 }  
 C.A[ C->n ] = 0; //Эквивалентно '\0'  
return C;  
}

Здесь в качестве одного из операндов выступает множество-универсум. Результат — элементы универсума, которых нет в исходном множестве.

Поскольку количество повторений цикла по элементам универсума постоянно, временная сложность операции — *O*(*n*). Однако, если учесть, что мощность универсума не может быть меньше мощности его подмножеств: |*U*|  ≥ *n*, более точной будет оценка *O*(|*U*| \* *n*), более пессимистическая по сравнению с *O*(*n*2).

Разумеется, нельзя обойтись без функции *Show*( ) для вывода множества на экран: это единственный способ увидеть результат обработки, поскольку сами множества из вызывающей программы недоступны.

void Set :: Show( ) { cout << ‘\n’ << S << " = [" << A << "]"; }

Для определения мощности множества нужна специальная функция *power*( ). Эта функция просто возвращает значение закрытой переменной *n*, в которой другие функции поддерживают значение текущей мощности множества. Поскольку функция не только объявлена, но и определена внутри класса, она по умолчанию является *встроенной.* К объявлению функции неявно добавляется спецификатор *inline*. Это означает, что никакой функции не создаётся, вместо этого в каждую точку вызова просто подставляется значение закрытой переменной *n*. Таким образом, запрет на доступ к *n* обходится без всяких дополнительных расходов на вызов функции.

Все функции-члены, кроме перечисленных ранее, объявлять и определять не обязательно, компилятор делает это автоматически. Но в данном случае автоматически определяемые функции не подходят.

Так, при создании объекта класса *Set* выделяется память, после чего вызывается функция-конструктор *Set*( ). По умолчанию эта функция — пустая, она ничего с памятью не делает, и использовать такой объект невозможно. Поэтому конструктор надо определить явно. Сделаем так, чтобы он создавал пустое множество латинских букв, представленное строкой символов:

Set :: Set( ): n(0), S ('0'), A(new char[ N+1 ]) { A[ 0 ] = 0; }

В этом примере одни переменные инициализируются в заголовке, другие — в теле конструктора. Оба способа можно комбинировать произвольным образом, но нужно учитывать, что порядок инициализации переменных полностью определяется порядком их объявления в классе и не может быть изменён. В данном примере значение переменной *N* используется для создания строки *A*, поэтому важно, что переменная *N* объявлена в классе раньше, чем указатель *A*. Массив символов объявляется на 1 символ длиннее мощности универсума, чтобы резервировать место под ограничивающий нуль. Поскольку строка должна быть пустой, ограничивающий нуль записывается в её начало. Инициализировать остальную часть массива не обязательно.

Если требуется иметь несколько способов создания объекта, для каждого способа объявляется свой конструктор, отличающийся от других типом и/или количеством аргументов. В примере объявлен конструктор с одним символьным аргументом. Это может быть конструктор, генерирующий случайную строку латинских букв. Аргумент используется для инициализации тега. С помощью датчика случайных чисел генерируется *N* битов, и для каждого единичного бита соответствующий ему элемент добавляется в множество. Одновременно подсчитывается фактическая мощность множества *n*. По окончании генерации в строку добавляется ограничитель. Сгенерированное множество выводится на экран.

Set :: Set(char s): S(s), n(0), A(new char[ N+1 ])  
{  
 for (int i = 0; i < N; i++)  
 if (rand() % 2) A[ n++ ] = i + 'A';  
 A[n] = 0;  
 cout << ‘\n’ << S << " = [" << A << "]";  
}

Следующие две функции-члена — конструктор копирования и перегрузку присваивания — для класса-значения обычно не определяют. Обе эти функции по умолчанию копируют один объект в другой по принципу «байт в байт». Если ничего другого не требуется, определять эти функции не нужно, так как компилятор наверняка сделает это лучше.

В данном же случае такое копирование не годится, потому что в классе есть указатель на дополнительную память, и копирование приведёт к тому, что указатели *A* в обоих объектах будут указывать на одну и ту же строку.

Конструктор копирования имеет единственный аргумент — константную ссылку на объект того же класса. Определить конструктор можно так:

Set :: Set(const Set & B) : S(B.S), n(B.n), A(new char[N+1])  
{ char \*s(B.A), \*t(A);  
 while(\*t++ = \*s++);  
}

Здесь переменные *S* и *n* копируются обычным способом, а для указателя *A* выделяется память под новую строку, и в неё затем копируется содержимое старой. Переменную *N* копировать не нужно: она является общей для всех экземпляров объекта типа Set.

Функция-член для перегрузки присваивания отличается от копирования тем, что объект в левой части оператора уже существует. Более того, он может совпадать с аргументом (самоприсваивание). Поэтому первое, что функция должна сделать — проверить это. Затем текущий объект уничтожается и создаётся новый. В нашем случае можно ограничиться переносом содержимого строки в имеющуюся память. Поскольку результат операции присваивания может быть использован в выражении, например в цепочке присваиваний, функция должна возвращать значение объекта, для которого она вызвана. Это делается с помощью встроенного указателя *this*. Копировать тег нет смысла: устанавливается некоторое условное значение «*R*» (от слова «*result*»).

Set & Set :: operator = (const Set& B)  
{ if (this != &B)  
 { char \*s(B.A), \*t(A); n = B.n; S = 'R';  
 while(\*t++ = \*s++); }  
 return \*this;   
}

Конструктор копирования используется при инициализации объекта содержимым другого объекта в момент объявления, а также при передаче аргумента в функцию как параметра по значению и при возврате объекта как результата работы функции. Функция перегрузки присваивания вызывается соответствующим оператором. Бывают ситуации, когда эти функции в программе не нужны. Чтобы исключить трудно выявляемую ошибку в программе из-за использования функций по умолчанию, рекомендуется объявить ненужные функции в закрытой части класса. Можно даже не определять их. Невольное создание в программе ситуации, когда такая функция вызывается, будет ошибкой, выявляемой компилятором.

В стандарте С++11 добавлены конструктор копирования и перегрузка присваивания в варианте «**с переносом**». Они используются, если источником данных является временный объект. Вместо того, чтобы создавать копию данных, принадлежащих объекту, в варианте «с переносом» эти данные просто передаются объекту-приёмнику:

Set :: Set(Set && B)   
 : S(B.S), n(B.n), A(B.A) //Копирование с переносом  
{ B.A = nullptr; }  
Set & Set :: operator = (Set&& B) //Присваивание с переносом  
{ if (this != &B)  
{ n = B.n; A = B.A; S = 'R'; B.A = nullptr; }  
 return \*this;   
}

Указатель на строку в объекте-источнике обнуляется, чтобы сделать удаление этого объекта безопасным.

Последняя функция-член в объявлении класса — это деструктор, который автоматически вызывается при уничтожении объекта. В нём указано дополнительное действие, которое нужно выполнить перед освобождением памяти из-под объекта: уничтожить строку *A*. Поскольку деструктор определён внутри класса, он, как и *power*( ), тоже является встроенной функцией. Деструктор вызывается явно в операторе *delete* или неявно — при выходе из блока, в котором объект был определён. Объекты уничтожаются в порядке, обратном порядку их создания.

**Программа, использующая объекты класса *Set*** (программа-клиент), может выглядеть так:

#include <string>  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <iostream>  
using namespace std;  
#include "Set.h" //Модуль с определением класса и его функций  
int Set ∷ N = 26; // определение статического члена класса  
const long q0 = 100000; // количество повторений цикла времени  
int main( )  
{ srand(time(nullptr));  
 Set A('A'), B('B'), C('C'), D('D'), E;  
 clock\_t begin = clock( );  
 for(long q =0; q < q0; ++q)  
 { E = (A | B) & (C & ~D); }  
 clock\_t end = clock( );  
 E.Out( );  
 cout << " Middle power =" <<  
 (A.int() + B.int( ) + C.int( ) + D.int( ) + E.int( )) / 5 <<  
 " Time=" << end – begin<< " / " << q0 << endl;  
 cin.get( );  
 return 0;  
}

В программе определяются пять множеств. Для исходных множеств *A*, *B*, *C* и *D* используется конструктор, генерирующий случайное множество с выводом на экран результата. Множество *E* генерируется конструктором по умолчанию как пустое. Затем множество вычисляется с использованием перегруженных операций, и результат выводится на экран. Далее вычисляются и выводятся средняя мощность всех множеств и время решения задачи.

Объявление класса *Set* и определения всех функций-членов находятся в подключаемом модуле *Set.h*. Определение статического члена — переменной *N* — помещено сразу после модуля. На самом деле оно является его частью. В самой программе никакой информации об устройстве модуля *Set.h* не имеется. Чтобы использовать другой способ хранения множеств в памяти, достаточно просто подменить модуль.

Вычисление множества *E* выполняется одним оператором присваивания, как в варианте для машинных слов. Но временная сложность этого вычисления не будет константной, она определяется функциями, реализующими операции над множествами и, следовательно, по-прежнему зависит от способа их хранения в памяти.

*Контрольные вопросы.*

1. Можно ли использовать ключевое слово *struct* в том же смысле, что и *class*?

class MyClass {   
 int A;  
public:  
int FunA ( );  
};  
int FunB( MyClass A );  
Может ли функция *FunA*( ) использовать переменную *A*, и если «да», то каким образом?

1. Тот же класс, сигнатура функции теперь   
   int FunB( const MyClass & S );  
   Каким образом функция *FunB*( ) может использовать переменную *A* объекта *S*?
2. Каким образом функция *FunB*( ) может использовать функцию *FunA*( ) объекта *S* в примере выше?
3. class MyClass {   
    int A;  
   public:  
   MyClass ( int );  
   };  
   Можно ли создать массив объектов типа *MyClass*?
4. class MyClass {   
    static int A;  
   public:  
   MyClass ( int );  
   };  
   Что означает атрибут *static* у переменной-члена *A* класса *MyClass*?
5. Как следует задавать значение статической переменной *A* в примере выше?
6. С каким ключевым словом следует объявить переменную в классе, чтобы она была общей для всех экземпляров объекта соответствующего типа?
7. С каким ключевым словом следует объявить функцию, чтобы она, не будучи членом класса, имела доступ ко его закрытым полям?
8. Можно ли объявить класс, который будет играть роль элемента множества списка и множества в целом?
9. Что лучше — объявить класс-элемент списка как структуру внутри класса-списка или объявить два отдельных класса, связав их дружбой?
10. К какому типу классов относится следующий класс?

template <class T> class less {  
 public:  
bool operator( )(T a, T b) { return a < b; }  
};

1. К какому типу классов относится следующий класс?

template <class T, int p = 26> class MyClass {  
 enum( N = p);  
 int n;  
 T Data[N];  
public:  
 MyClass( ): n(0) { Data[0] = 0; }  
/…  
};

1. К какому типу классов относится следующий класс?

template <class T, int p = 32> class MyClass {  
 enum( N = p);  
 int n;  
 T \* Data;  
public:  
 MyClass( ): n(0), Data(new T[N]) { Data[0] = 0; }  
/…  
};

1. К какому типу классов относится следующий класс?

template <class T, int p = 26> class MyClass {  
 enum( N = p);  
 int n;  
 vector< T > Data;  
public:  
 MyClass( ): n(0) { }  
/…  
};

1. К какому типу классов относится следующий класс?

template <class T, int p = 26> class MyClass {  
 enum( N = p);  
 int n;  
 unique\_ptr< T[ ] > Data;  
public:  
 MyClass( ): n(0), Data(make\_unique(new T[N]) { }  
/…  
};

1. Каковы преимущества объектно-ориентированного подхода к програм­миро­ва­нию по сравнению с процедурным?

# 8. Перегрузка операций

СПОСОБ: объявить функцию с именем «operator #», где за ключевым словом operator следует знак перегружаемой операции.

— Позволяет использовать более удобные средства работы с объектами, чем обычные функции.

Пример:

E = A | (B & C & ~D);

вместо E = or(A, and(B, and(C, not(D))));

или not(D, E); and(C, E, E); and(B, E, E); or(A, E, E);

ПРАВИЛА перегрузки операций:

— перегружать можно почти все существующие в языке знаки операций. Исключения: «.», «::», «?:», «#», sizeof;

— нельзя придумывать новые знаки операций;

— нельзя переосмысливать операции над встроенными типами: хотя бы один аргумент должен быть пользовательским;

— нельзя изменить количество аргументов (1 или 2) и применять аргументы по умолчанию. Но можно в двуместной операции игнорировать один из аргументов;

— нельзя изменить приоритеты и ассоциативность.

Пример: перегружены операции «+» и «\*» и предложен знак «^» для возведения в степень комплексных чисел. Выражение *a* = *b* + *c*\**d*^*n* означает *a* = (*b* + (*c*\**d*))^*n*, а не *a* = ( *b* + (*c*\*(*d*^*n*)), как ожидал программист.

ВЫВОД: перегрузка должна быть естественной. Цель: улучшить читабельность текста программы, а не наоборот (показать крутизну программиста)! Если эта цель не достижима, от перегрузки следует отказаться.

СЛЕДСТВИЕ: нужно по возможности поддерживать семантику перегружаемых операций, чтобы можно было строить выражения привычным способом. А именно: результат операции должен быть пригоден как аргумент для следующих операций.

Можно ли отказаться от перегрузки совсем? Не получится:

— перегрузка присваивания определена для пользовательского типа (равно как и взятие адреса, запятая);

— перегрузка применяется в стандартных библиотеках:

«+» для *string* — конкатенация строк;

сдвиги — для объектов *iostream*.

Естественно применять арифметические операции для комплексных, рациональных, сверхдлинных чисел и т. п. (и это делается в доступных библиотеках).

— СПОСОБЫ объявления: функция-член, функция-друг, обычная функция. Какой способ выбрать?

**Обычная функция** возможна, если не требуется доступ к закрытой части класса.

**Операция, манипулирующая адресом объекта**, должна быть (нестатическим) членом: *operator*=, *operator*[ ], *operator*( ), *operator*->

**Операции** присваивание, взятие адреса и запятая обладают предопределёнными свойствами для объектов. Их можно сделать частными (запретить) или определить иначе.

**Операция**, первый аргумент имеет стандартный (другой существующий) тип, не может быть членом.

*Пример*: вывод множества на экран (первый аргумент — поток):

class Set {  
 const int U = 26;  
 int n; char S; char A[U + 1];  
 friend ostream& operator <<(ostream & o, const set& src)   
 { o << S << '=' << A << std::endl; return o; }   
//…  
}

Для данных *int* *i*++ означает то же, что и *i* += 1 или *i* = *i* + 1. Для пользовательских типов такой связи по умолчанию нет, все варианты надо определить явно.

ПРЕОБРАЗОВАНИЕ ТИПА

— конструктор с одним аргументом по существу является преобразователем типа и может вызываться неявно. Чтобы это запретить, его описывают с ключевым словом *explicit*.

explicit Set(int power); // Конструктор случайного множества заданной мощности.

Преобразование в стандартный или существующий тип можно определить явно:

*operator* *int* (*const* *Set* &) { *return* *n*; } — преобразование к целому. Возможный смысл — выдача мощности для множества.

Функция не содержит указания на тип возвращаемого значения!

Во избежание недоразумений для выдачи мощности лучше использовать вариант с явным именем функции: *Set*::*power*( ) { *return* *n*; }

ВОЗВРАТ ССЫЛКИ

Обязателен, если результат может быть слева от присваивания (=, [ ], ->).

Разумен, если возвращается аргумент-ссылка (как в перегрузке operator >>) или аргумент-объект (как в перегрузке присваивания):

Set& Set :: operator = (const Set & B)  
{ if (this != &B) {  
 n = B.n; S = B.S;  
 char \*a(A), \*b(B.A); while(\*a++=\*b++); //); ctrcpy (A, B.A);   
 }   
 return this\*  
}

ИНКРЕМЕНТ и ДЕКРЕМЕНТ — примеры для работы с множествами

1. Инкремент для объекта — указателя на элемент списка

Y\* X::operator ++ ( ) //префиксный инкремент для объекта-указателя  
{ p = p->next; //  
 return p;  
}  
Y\* X::operator ++ (int) //то же — постфиксный  
{ X temp (\*this);  
 ++ \*this;  
 return temp;  
}

1. Инкремент для множества-массива: увеличение мощности.

const Set & Set::operator ++ ( ) const //префиксный вариант  
{  
 if(n < N) for (char c = 'A'; c <= 'Z'; ++c) //Цикл по универсуму  
 { bool f = true;  
 for (El \* j = A; j && f; j = j->next) //Добавить отсутствующий  
 if(c == j->e) f = false;  
 if (f){ A = new El(c, A) , ++n; break; }  
 }  
 return \*this;  
}

Постфиксный вариант совпадает с предыдущим примером.

Комбинированные присваивания

Нужно обязательно объявлять оба варианта — с присваиванием и без. Разумно обеспечить согласованность: двуместную операцию определить через комбинированное присваивание.

Пример для множества на основе списка:

const Set & Set::operator &= (const Set & B)  
{ Set C;  
// std::swap(C, \*this); «Естественный» способ приводит к бесконечной рекурсии!  
std::swap(C.A,A); std::swap(C.S,S); std::swap(C.n,n); //Вместо *Set* *C*(\**this*);  
 for (El \* i = C.A; i; i = i->next)  
 for (El \* j = B.A; j; j = j->next)  
 if (i->e == j->e)  
 A = new El(i->e, A), ++n;  
 }  
 return \*this;  
}  
Set Set::operator & (const Set& B) const  
{ Set C(\*this); return (C &= B); }

**Перегрузка операций** — естественный способ расширения языка в нужную предметную область.

**Цель**: получить эффективный и удобный в сопровождении программный код.

Перегружать можно почти всё.

Нельзя:

— придумывать новые знаки операций;

— изменить количество аргументов (один или два);

— изменить приоритет.

**Полный перечень операций языка С++,   
упорядоченный по уменьшению приоритета**

(**выделены операции**, не пригодные для перегрузки)

1. **∷ —** **уточнение области действия**
2. круглые скобки:  
    (выражение) — группировка для изменения приоритета  
    имя( ) — вызов функции (слева направо)  
    тип(выражение) — конструкция значения  
    [ ] — индексация  
    **. —** **прямое членство**  
    -> — косвенное членство  
    ++ — постфиксный инкремент  
    – – — постфиксный декремент  
    **const\_cast, dynamic\_cast, reinterpret\_cast, static\_cast, typeid — снятие атрибута *const*, преобразование типа, взятие идентификатора типа**
3. ! ~ + – — одноместные «не», обратный код, плюс, минус  
    ++ – – префиксные инкремент и декремент  
    & — взятие адреса  
    \* — разыменование  
    (тип) выражение — приведение типа (в стиле Си)  
    **sizeof — взятие размера** alignof, new, new[ ], delete, delete [ ], noexcept — взятие выравнивания, выделение и освобождение памяти, атрибут отсутствия исключений
4. **.\* — разыменование члена — прямое**–>\* — то же, косвенное
5. 5. \* / % — арифметические: умножение, деление, остаток от деления

6. + – — арифметические: сложение, вычитание

7. ≫ ≪ — сдвиги

8. < <= >= > — сравнения на меньше/больше

9. == != — проверка на совпадение

10. & — поразрядное «И»

11. ^ — исключающее «ИЛИ»

12. | — поразрядное «ИЛИ»

13. && — логическое «И»

14. || — логическое «ИЛИ»

15. **: ? — условная операция**

16. = \*= /= %= += -= &= ^= |= ≪= ≫= — присваивания

17. throw — исключение

18. , — запятая

ВАЖНО: использование перегрузки операций — то же самое, что и вызовы соответствующих функций. Запись

E = A | (B & C & ~D);

для перегруженных операций на самом деле эквивалентна

E = A.operator | (B.operator & (C.operator &(D.operator ~( ));

Принципиальное отличие: аргументы функций вычисляются всегда и порядок их вычисления не определён.

Следствие: не рекомендуется во избежание недоразумений перегружать логические операции и запятую.

К перегрузкам можно отнести и преобразование типа. Конструктор с одним аргументом — преобразователь типа аргумента в тип объекта.

*Set*(10) — конструктор множества мощностью 10 = преобразователь целого числа в множество. Предохранение:

explicit Set(int);

Преобразование объекта в стандартный (любой другой) тип:

*operator int*( ) — функция-член для преобразования (множества) в целое. Не рекомендуется, потому что смысл не очевиден!

В общем случае: автоматическое преобразование типа объекта в месте, где ожидается другой тип (аргумент функции). Альтернатива — **явные преобразования**

— преобразование «в стиле Си» — (тип)выражение;

— альтернатива для С++ — тип(выражение);

**Рекомендуемый способ** — *static\_cast*<тип>(выражение) или reinterpret\_cast<тип>(выражение).

Первый способ — это преобразование значения выражения к указанному типу, например, целого к вещественному. Второй способ означает замену типа без преобразования выражения, например, объявление указателя целым числом или наоборот.

**Специальные способы** (потенциально опасные!):

const\_cast — снятие атрибута const/volatile,

dynamic\_cast — преобразование указателя в иерархии типов;

**Перегрузка круглых скобок**: класс-функтор, используемый обычно как аргумент в качестве указателя на функцию:

class less{  
 int A;  
public:  
 less(int a): A(a) { }  
 bool operator( )(int B){return B < A;}  
};

Использование:

erase(arr, N, less(10)); //Удалить из массива элементы, меньшие 10.

**Перегрузка присваивания** — встроена в язык. Объявление класса автоматически означает объявление для него в качестве функции-члена перегрузки присваивания.

**Типовое использование** перегрузки операций:

— круглые скобки: функторы;

— арифметические: (комплексные, сверхдлинные числа);

— поразрядные (работа с множествами);

— сравнения (предикаты — функции с результатом типа bool);

— операции с указателями: умные указатели, итераторы;

— префиксный и постфиксный инкремент/декремент.

ВАЖНО. Префиксную и постфиксную формы категорически рекомендуется определять одновременно.

Пример:

Y\* X::operator ++ ( ) //префиксный инкремент для объекта-указателя  
{ p = p->next; //(Вариант для списка: инкремент перемещает указатель)  
 return p;  
}  
Y\* X::operator ++ (int) //то же — постфиксный  
{ X temp (\*this);  
 ++ \*this; //Постфиксный реализован через префиксный !  
 return temp;  
}

ВАЖНО. Согласованность определений перегрузки для логически связанных операций.

Пример: двуместная операция и комбинированное присваивание.

const Set & Set::operator &= (const Set & B)  
{ Set C;  
 std::swap(C.A,A); std::swap(C.S,S); std::swap(C.n,n); //вместо *Set* *C*(\**this*);  
 for (El \* i = C.A; i; i = i->next)  
 for (El \* j = B.A; j; j = j->next)  
 if (i->e == j->e)  
 A = new El(i->e, A), ++n;  
 }  
 return \*this;  
}  
Set Set::operator & (const Set& B) const  
{ Set C(\*this); return (C &= B); }

**Класс-указатель**

Для обычных указателей эквивалентны *p*->*m* == (\**p*).*m* == *p*[0].m

Для пользовательских объектов-указателей это нужно обеспечить:

template < class Y >  
class X { //Пример класса-указателя с полным набором операций с ним  
 Y\* p;  
public:   
 Y\* operator -> ( ) { return p; }  
 Y& operator \* ( ) { return \*p; }  
 Y& operator [ ] (int i) { return p[ i ]; }  
 Y\* operator ++ ( )  
 Y\* operator ++ (int)  
};

Особый случай: класс для множества на основе списка.

Совместное использование двух классов-контроллеров:

— элемент списка;

— список в целом.

Вариант реализации:

class El{  
 char e;  
 El \*next;  
 El(char e, El \*n = nullptr): e(e), next(n) { }  
 ~El(){delete next;}  
 El(const El &) = delete;  
 El & operator=(const El &) = delete;  
 El(El &&) = delete;  
 El & operator=(El &&) = delete;  
friend class Set;  
};  
class Set  
{  
private:  
 static const int N; //Мощность универсума  
 int n; //Мощность множества  
 char S; //Тег  
 El \*A; //Список элементов  
public:  
 Set(); //Пустое множество  
 Set(char); //Случайное произвольной мощности (аргумент игнорируется)  
 Set(const Set &);  
 Set(Set &&);  
 Set & operator = (const Set&);  
 Set & operator = (Set &&);  
 ~Set() { delete A; }  
 void Show();  
 int power() { return n; }  
//…  
};

Альтернативный вариант:

class Set  
{  
private:  
 static const int N; //Мощность универсума  
struct El{  
 char e;  
 El \*next;  
 El(char e, El \*n = nullptr): e(e), next(n) { }  
 ~El(){delete next;}  
 };  
 int n; //Мощность множества  
 char S; //Тег  
 El \*A; //Список элементов  
public:  
 Set(); //Пустое множество  
//…  
};

**Парадоксальный вариант**: в объявлении класса *Set* убираем звёздочку в объявлении поля *A*.

class Set  
{  
private:  
 int n; //Мощность множества  
 El A; //Список элементов = полноценный тип  
public:  
 Set(); //Пустое множество  
 Set(char); //Случайное произвольной мощности (аргумент игнорируется)  
// Set(const Set &);  
// Set(Set &&);  
// Set & operator = (const Set&);  
// Set & operator = (Set &&);  
// ~Set() { delete A; }  
 void Show();  
 int power() { return n; }  
//…  
};

Список в таком варианте всегда имеет один элемент — головной, который содержится в *Set*; его поле для символа можно использовать под тег, а нулевое значения указателя *next* означает пустое множество.

Класс *Set* превратился в класс-значение, и ему нужны только конструкторы, а класс *El* — в полноценный класс-контроллер, и ему нужен полный набор служебных функций.

Эти функции автоматически используются при копировании, присваивании, уничтожении объекта типа *Set* без дополнительных указаний от программиста. Иногда (в сложных, сомнительных случаях) пишут:

Set(const Set &) = default;  
 Set(Set &&) = default;  
//…

Существо изменения: отношение дружбы между классами *EL* и *Set* заменено отношением «содержит».

# 9. Перехват управления памятью: перегрузка *new* и *delete*

Специфика работы со списком: необходимость поэлементного заказа/освобождения памяти. Универсальная операция == дорогая‼!

Альтернатива: один раз выделить память под списки и самому управлять ею на уровне элементов. Способ: перегрузка *new* и *delete* для элемента списка.

Перегруженные *new* и *delete* — всегда статические функции-члены.

**Пример программы для эксперимента** с отслеживанием времени жизни объектов при работе с множествами-списками

1. Объявление класса «элемент списка» с перехватом управления свободной памятью (перегрузка операций new и delete)

class El{  
 char e;  
 El \*next;  
 static const int maxmup = 300;  
 static El mem[maxmup]; //Свободная память для элементов списков  
 static int mup, mup0;  
public:  
 El(): e('!'), next(nullptr){ }  
 El(char e, El \*n = nullptr): e(e), next(n) { std::cout << "+" << e; }  
 ~El(){ if(this) { //Проверка обязательна  
 if(next) { delete next;}  
 std::cout << "-" << e;  
 e = '\*'; }  
 else cout << "<Пусто!>";  
 }  
 static void\* operator new(size\_t) {  
 return (mup < maxmup? &mem[mup++] : nullptr); }  
 static void operator delete(void \*, size\_t) { }  
 static void mark(){ mup0 = mup;} //Фиксировать состояние памяти  
 static void release() { mup = mup0; } //Сбросить до фиксированного  
 static void clear(){ mup = 0; } //Очистить память полностью  
 friend class Set;  
 friend std::ostream & operator << ( std::ostream & o, El & S);  
 friend static void memOut();  
};  
std::ostream & operator << (std::ostream & o, El & S)  
{  
 for (El\* p = &S; p; p = p->next) o << p->e;  
 return o;  
}

El El::mem[El::maxmup]; //"Свободная память"  
int El::mup = 0, El::mup0 = 0;

void memOut( ) //Вывод использованного содержимого "свободной памяти"  
{  
 std::cout << "\n\nПамять элементов списков (всего - " << std::dec << El::mup << ")\n";   
 for(int i = 0; i < El::mup; ++i) cout << El::mem[i].e;   
}

2. Объявление класса «множество-список» с автоматической нумерацией вновь создаваемых множеств

class Set  
{  
private:  
 const int N = 26; //Мощность универсума  
 static int num; //Порядковый номер множества  
 int n; //Мощность множества  
 char S; //Тег  
 El \*A; //Список элементов

public:  
 Set(); //Пустое множество  
 Set(char); //Случайное произвольной мощности (аргумент игнорируется)  
 Set(const Set &);  
 Set(Set &&);  
 Set & operator = (const Set&);  
 Set & operator = (Set &&);  
 ~Set() {   
 std::cout << "Удалено " << S <<"(" << std::dec << n << ") = [" << \*A << "]";  
 A->El::~El(); cout << std::endl; } //Здесь нужен явный вызов деструктора

void Show();  
 int power() { return n; }  
 void swap(Set & other) { std::swap(S, other.S); std::swap(n, other.n); std::swap(A, other.A);}  
 Set & operator |= (const Set&);  
 Set & operator &= (const Set&);  
 Set operator | (const Set&) const;  
 Set operator & (const Set&) const;  
 Set operator ~ () const;  
};

3. Определение функций-членов класса set с отслеживанием появления  
и уничтожения множеств и главная программа

Set::Set() : n(0), S('A'+num++), A(nullptr)  
{ std::cout << "-> Создано " << S << "(" << n << ") = [" << \*A << "] \n";}

Set::Set(char) : S('A'+num++), n(0)  
{  
 A = nullptr;  
 for (int i = 0; i < N; i++)  
 if (rand() % 2)  
 A = new El(i + 'A', A), ++n;  
 std::cout << "-> Создано " << S << "(" << n << ") = [" << \*A << "] \n";  
}

Set::Set(const Set & B) : n(B.n), S('A'+num++), A(nullptr)  
{  
 for(El \* p = B.A; p; p = p->next) A = new El(p->e, A);  
 std::cout << "-> Создано " << S << "(" << n << ") = [" << \*A << "] из " << B.S << std::endl;  
}

Set::Set( Set && B) : n(B.n), S('A'+num++), A(B.A)  
{  
 B.A = nullptr;  
 std::cout << "-> ПРИНЯТО " << S << "(" << n << ") = [" << \*A << "] из " << B.S << std::endl;  
}

Set & Set::operator &= (const Set& B)   
{  
 Set C;  
 for (El \* i = A; i; i = i->next)  
 {  
 for (El \* j = B.A; j; j = j->next)  
 if (i->e == j->e)  
 C.A = new El(i->e, C.A), ++C.n;  
 }  
 swap(C);  
 std::cout << "; Получено " << S << "(" << n << ") = [" << \*A << "] = " << C.S << "&" << B.S << std::endl;  
 return \*this;  
}  
Set Set::operator & (const Set& B) const  
{  
 Set C(\*this);  
 std::cout << "Вычисление " << C.S << " & " << B.S << std::endl;  
 return C&=B;  
}

Set & Set::operator |= (const Set & B)   
{  
 Set C(\*this);  
 for (El \* i = B.A; i; i = i->next)  
 {  
 bool f = true;  
 for (El \* j = A; f && j; j = j->next)  
 f = f && (i->e != j->e);  
 if (f)  
 C.A = new El(i->e, C.A), ++C.n;  
 }  
 swap(C);  
 std::cout << "; Получено " << S << "(" << n << ") = [" << \*A << "] = " << C.S << "|" << B.S << std::endl;  
 return \*this;  
}  
Set Set::operator | (const Set& B) const  
{  
 Set C(\*this);  
 std::cout << "Вычисление " << C.S << " | " << B.S << std::endl;  
 return C|=B;  
}  
Set Set::operator ~ ( )const  
{  
 Set C;  
 for (char c = 'A'; c <= 'Z'; ++c)  
 {  
 bool f = true;  
 for (El \* j = A; j && f; j = j->next)  
 if(c == j->e) f = false;  
 if (f)  
 C.A = new El(c, C.A) , ++C.n;  
 }  
 std::cout << "; Получено " << C.S << "(" << C.n << ") = [" << \*C.A << "] = ~" << S << std::endl;  
 return C;  
}

Set& Set::operator = (const Set & B)  
{  
 if (this != &B)  
 { std::cout << "\nУдалено " << S << "(" << n << ") = [" << \*A << "]";  
 delete A;  
 A = nullptr;  
 n = 0;  
 for(El \* p = B.A; p; p = p->next)   
 A = new El(p->e, A), ++n;  
 S = 'A'+num++;  
 }  
 std::cout << "; Создано " << S << "(" << n << ") = [" << \*A << "] из " << B.S << std::endl;  
 return \*this;  
}

Set& Set::operator = (Set && B)  
{ std::cout << "\nУдалено " << S << "(" << n << ") = [" << \*A << "]";  
 swap(B);  
 delete B.A; B.A = nullptr;  
 S = 'A'+num++;  
 std::cout << "; ПЕРЕДАНО " << S << "(" << n << ") = [" << \*A << "] из " << B.S << std::endl;  
 return \*this;  
}

void Set::Show( )  
{  
 std::cout << '\n' << S << "(" << hex << A << ") = [ ";  
 for(El \* p = A; p; p = p->next) std:: cout << p->e << " ";  
 std::cout << "]\n\n";  
}

const int Set :: N = 26;  
int Set :: num = 0;  
  
int main()  
{  
 {  
 setlocale(0, "");  
 El::release();  
 Set A('A'), B('B'), C('C'), D('D');  
 cout << "\nРезультат:\n";  
 (Set(A) | Set(B) & Set(C) & ~Set(D)).Show( );  
 cout << "\n=== pause ===\n";  
 cin.get( );  
 }  
 memOut( );  
 cout << "\nВсё!\n=== pause ===\n";  
 cin.get();  
 return 0;  
}

4. Результат эксперимента

Можно отследить моменты появления и удаления как отдельных элементов, так и каждого множества в целом, в том числе и работу конструктора копии, копии с переносом, присваивания и комбинированных операций.

+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано A(13) = [XUSQPONMLKEBA]   
+A+D+E+G+I+K+L+M+O+P+R+S+U+V+W+Y-> Создано B(16) = [YWVUSRPOMLKIGEDA]   
+B+C+D+E+F+G+J+T+V+Z-> Создано C(10) = [ZVTJGFEDCB]   
+A+C+D+K+N+P+Q+U+V+W+X+Y-> Создано D(12) = [YXWVUQPNKDCA]   
  
Результат:  
+Y+X+W+V+U+Q+P+N+K+D+C+A-> Создано E(12) = [ACDKNPQUVWXY] из D  
-> Создано F(0) = [ ]   
+B+E+F+G+H+I+J+L+M+O+R+S+T+Z; Получено F(14) = [ZTSROMLJIHGFEB] = ~E  
-> ПРИНЯТО G(14) = [ZTSROMLJIHGFEB] из F  
Удалено F(14) = []<Пусто!>  
+Z+V+T+J+G+F+E+D+C+B-> Создано H(10) = [BCDEFGJTVZ] из C  
+Y+W+V+U+S+R+P+O+M+L+K+I+G+E+D+A-> Создано I(16) = [ADEGIKLMOPRSUVWY] из B  
+A+D+E+G+I+K+L+M+O+P+R+S+U+V+W+Y-> Создано J(16) = [YWVUSRPOMLKIGEDA] из I  
Вычисление J & H  
-> Создано K(0) = []   
+V+G+E+D; Получено K(4) = [DEGV] = J&H  
Удалено J(16) = [YWVUSRPOMLKIGEDA]-A-D-E-G-I-K-L-M-O-P-R-S-U-V-W-Y  
+D+E+G+V-> Создано L(4) = [VGED] из K  
Удалено K(4) = [DEGV]-V-G-E-D  
+V+G+E+D-> Создано M(4) = [DEGV] из L  
Вычисление M & G  
-> Создано N(0) = [ ]   
+E+G; Получено N(2) = [GE] = M&G  
Удалено M(4) = [DEGV]-V-G-E-D  
+G+E-> Создано O(2) = [EG] из N  
Удалено N(2) = [GE]-E-G  
+X+U+S+Q+P+O+N+M+L+K+E+B+A-> Создано P(13) = [ABEKLMNOPQSUX] из A  
+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано Q(13) = [XUSQPONMLKEBA] из P  
Вычисление Q | O  
+X+U+S+Q+P+O+N+M+L+K+E+B+A-> Создано R(13) = [ABEKLMNOPQSUX] из Q  
+G; Получено R(14) = [GABEKLMNOPQSUX] = Q|O  
Удалено Q(13) = [XUSQPONMLKEBA]-A-B-E-K-L-M-N-O-P-Q-S-U-X  
+G+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано S(14) = [XUSQPONMLKEBAG] из R  
Удалено R(14) = [GABEKLMNOPQSUX]-X-U-S-Q-P-O-N-M-L-K-E-B-A-G  
  
S(00182730) = [ X U S Q P O N M L K E B A G ]  
  
Удалено S(14) = [XUSQPONMLKEBAG]-G-A-B-E-K-L-M-N-O-P-Q-S-U-X  
Удалено P(13) = [ABEKLMNOPQSUX]-X-U-S-Q-P-O-N-M-L-K-E-B-A  
Удалено O(2) = [EG]-G-E  
Удалено L(4) = [VGED]-D-E-G-V  
Удалено I(16) = [ADEGIKLMOPRSUVWY]-Y-W-V-U-S-R-P-O-M-L-K-I-G-E-D-A  
Удалено H(10) = [BCDEFGJTVZ]-Z-V-T-J-G-F-E-D-C-B  
Удалено G(14) = [ZTSROMLJIHGFEB]-B-E-F-G-H-I-J-L-M-O-R-S-T-Z  
Удалено E(12) = [ACDKNPQUVWXY]-Y-X-W-V-U-Q-P-N-K-D-C-A  
  
=== pause ===  
Удалено D(12) = [YXWVUQPNKDCA]-A-C-D-K-N-P-Q-U-V-W-X-Y  
Удалено C(10) = [ZVTJGFEDCB]-B-C-D-E-F-G-J-T-V-Z  
Удалено B(16) = [YWVUSRPOMLKIGEDA]-A-D-E-G-I-K-L-M-O-P-R-S-U-V-W-Y  
Удалено A(13) = [XUSQPONMLKEBA]-A-B-E-K-L-M-N-O-P-Q-S-U-X

Память элементов списков (всего - 189)  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Всё!  
=== pause ===  
-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-!-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*

*Контрольные вопросы.*

1. Каким способом можно перегрузить операцию «|» для множеств — использовать функцию-член, функцию-друг, обычную функцию?
2. Каким способом можно перегрузить операцию «=»?
3. Каким способом можно перегрузить операцию «[ ]»?
4. Как различить перегрузку префиксного и постфиксного инкрементов «++»?
5. Можно ли при перегруженных «|» и «=» применять комбинированную операцию «|=»?
6. Какая сигнатура является корректной для двуместной операции объединения множеств?
7. Какая сигнатура является корректной для операции комбинированного присваивания — объединения множеств?
8. Какая сигнатура является корректной для перемещающего конструктора?
9. Можно ли ускорить выполнение оператора «*A* = *B* | *C*» с перегруженными операциями «=» и «|»?
10. Может ли функция-член для перегрузки операции возвращать ссылку на объект?
11. В каких случаях нужен перемещающий конструктор?
12. Как можно потребовать, чтобы следующая функция возвращала результат, используя перемещающий конструктор?

Set Set::And(const Set & rgt) const  
{ Set Result;  
 for(int i = 0; i < n; ++i)  
 for(int j = 0; j < rgt.n ++j) if(A[i] == rgt.A[i])  
 Result[Result.n++] = A[i];  
 return Result;  
}

1. Каким способом объект-множество лучше всего передавать в качестве аргумента в функцию?
2. В каком случае функция-член должна возвращать ссылку на объект?
3. Объявлен массив целых: *int* *A*[100]{0};

Какие элементы массива инициализируются нулём?

# 10. Понятие о шаблонах. Стандартная библиотека шаблонов

ШАБЛОНЫ — обобщённое программирование.

Алгоритм пригоден для разных (похожих) типов данных — написать шаблон.

Пример. Функция, меняющая местами содержимое своих аргументов.

template <class T>  
void swap(T a, T b) { T c(a); a = b; b = c; }

Функция пригодна для любого типа T, для которого определены конструктор копии и перегрузка присваивания.

Важно:

— объявление шаблона само по себе ничего не добавляет в код. Определение функции появится в коде только при использовании шаблона — своё для каждого использованного типа данных.

Пример.

int a=0, b=1; double f=0, p=3.14; Set A(«A»), B(«B»);  
swap<int>(a,b); swap(f,p); swap(A,B);

Здесь будут сгенерированы три варианта кода.

ШАБЛОННЫЙ КЛАСС — шаблонные все его функции-члены.

*Пример*: шаблонный класс «множество в массиве».

template <class T = char, unsigned N = 26, char s0 = 'A'>  
class set1{  
 int n;  
 T \*A;  
public:  
 set1( ) : n(0), A(new T[N+1]) { }  
 set1(int);  
};   
 template <class T = char, unsigned N = 26, char s0 = 'A'>  
set1 :: set1(int k) : set1( )   
{ for (int i = 0; i < N; ++i) if(rand( )%2) A[n++] = s0 + i;}

Аргументы шаблона — имена, указанные как *class* или *typename*, а также переменные (целые, указатели, ссылки) *const*, *constexpr* — вычисляемые на этапе компиляции.

Аргументы шаблона похожи на аргументы функции. Шаблон порождает программный код только по факту использования (инстанцирование шаблона).

При использовании шаблонного класса фактические аргументы шаблона обязательны (но могут использоваться значения по умолчанию).

Для шаблонных функций подстановка аргументов шаблона часто происходит автоматически (определяется из вызова функции).

Библиотека шаблонов — это набор исходных текстов (h-файлы), вставляемых в программу пользователя по мере надобности.

СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

Для популярных типов данных библиотека содержит определения КОНТЕЙНЕРОВ — классов, специфицируемых пользовательским типом данных, и шаблоны функций, реализующих популярные алгоритмы.

По отношению к пользовательскому классу контейнер находится в отношении «содержит». Пользовательский класс должен иметь набор служебных функций, удовлетворяющий требованиям контейнера.

Концепция Стандартной библиотеки шаблонов:

КОНТЕЙНЕР (итератор)→ АЛГОРИТМ →(итератор) КОНТЕЙНЕР

Контейнеры используются как источник данных и хранилище для результатов вычислений. Для связи контейнеров с алгоритмами используются особые интерфейсные объекты — итераторы. Это вспомогательные классы, обобщающие понятие указателя.

*Пример*: работа с календарными датами: чтение из файла, корректировка, запись результата.

class Date{ //Класс «календарная дата» с перегруженным вводом/выводом  
 //…  
 public:  
 istream& operator >>( istream& Date&);  
 ostream& operator <<(ostream&, Date&);  
 };

int main( )  
 { vector <Date> e; //Рабочая память  
 **copy**(istream\_iterator<Date>(cin),   
 istream\_iterator<Date>( ), back\_inserter(e)); //Чтение набора дат  
 vector<Date>∷iterator f = **find**(e.begin( ), e.end( ), «16.01.16»);  
 if(f != e.end( )) \*f = **actual\_date**( ); //Замена искомой даты на текущую  
 **sort**(e.begin(), e.end()); //Упорядочение  
  **unique\_copy**(e.begin(), e.end(), ostream\_iterator<Date>(cout, «\n»));  
 //Вывод результата с исключением дубликатов  
 }

В примере — единообразный подход к работе с последовательностями в памяти, во входном и выходном файлах. Алгоритм *copy* читает данные из стандартного потока с помощью итератора ввода и помещает их в контейнер *vector*, используя итератор вставки. Далее в векторе алгоритмом *find* ищется заданная дата. Используется прямой итератор. Возвращается значение итератора, указывающее на искомую дату или на конец данных. Если дата найдена, в месте, указанном итератором, происходит её замена на актульную.

Далее с помощью алгоритма *sort* даты упорядочиваются, а алгоритмом *unique\_copy* копируются в выходной поток. В последнем случае используется итератор вывода.

Библиотека STL — основные элементы:

— **контейнеры**;

— аллокаторы;

— **итераторы**: входные, выходные, прямые, двунаправленные, произвольного доступа; потоковые.

— **функторы** (plus, minus, multiplies, divides, modulus, negate) <functional>

— предикаты (less, equal\_to, greather\_equal, logical\_and/or/not)

— **алгоритмы** (includes, set\_intersection, set\_difference, set\_symmetric\_difference, set\_union, make\_heap, pop\_heap, push\_heap, sort\_heap, for\_each, transform, unique, copy, fill, generate, reverse, rotate, random\_shuffle, binary\_search, merge, sort, stable\_sort)

# 11. Контейнерные классы

Последовательные контейнеры: *vector*, *list*, *deque*, *array*, *forward\_list*

Адаптеры *stack, queue, prioriry\_queue*.

Ассоциативные контейнеры (множество и отображение): *map, multimap, set, multiset, unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap*.

Последовательные контейнеры и адаптеры впредь рекомендуются для работы с последовательностями, ассоциативные — откладываются до весны.

Объявление контейнера (на примере контейнера *vector*):

template <class T, class Allocator = allocator<T>>

class **vector** {…};

Обобщение для массива. Отличие: знает свой размер и никогда (при правильном использовании) не переполняется. При объявлении по умолчанию — пуст, заполняется в процессе работы. Является памятью прямого доступа, т.  е. поддерживает индексирование. Есть функция at(int) — индексирование с проверкой корректности индекса.

vector <char> A; //Объявление пустого вектора (расширение — с конца).  
vector <char> A(N, 0); //Вектор из *N* элементов, заполненный нулями.

Использование последнего вектора — такое же, как и обычного массива из *N* элементов.

В процесс выделения памяти можно вмешаться:

A.resize(N, 0); //Изменить размер на указанный, заполнить нулями.  
A.reserve(N); //Резервировать память не менее чем под *N* элементов.  
A.capacity( ); //Проверить текущий запас памяти.  
A.shrink\_to\_fit( ); //Сбросить лишнюю память (С++11).

Аргумент шаблона *Allocator* — подключение своего альтернативного средства работы с памятью.

Если не нужна свободная память — упрощённый вариант (С++11):

**array**<char>(N, 0);

Это неперемещаемый массив с фиксированным размером, который должен быть указан.

Специальная альтернатива для массива символов — *string, wstring*.

Контейнер ***list*** — двунаправленный список. Обеспечивает константную вставку в начало и в середину, обмен частями между несколькими списками.

***forward\_list***— упрощённая однонаправленная версия списка (С++11).

***deque*** — гибрид массива и списка: список из экстентов. Допускает константную вставку в начало и индексацию. База для очереди и стека.

ПОЛЯ КОНТЕЙНЕРОВ (стандарт для использования в программах):

*value\_type* — тип элемента  
 *key\_type* — тип ключа (для ассоциативных)   
 *key\_compare* — тип критерия сравнения (для ассоциативных)  
 *size\_type* — тип индекса, счётчика и т. п.  
 *iterator* — итератор (указатель на элемент)  
 *const\_iterator* — константный итератор (указатель на константу)  
 *reverse\_iterator* — обратный итератор  
 *const\_reverse\_iterator* — константный обратный =  
 *reference* — ссылка на элемент  
 *const\_reference* — константная ссылка на элемент

ФУНКЦИИ ДЛЯ ПРОСМОТРА (установка на начало/конец)

*iterator begin* ( ); //установка на первый элемент  
 *const\_iterator сbegin* ( ) *const*;  
 *iterator end* ( ); // — на следующий за последним  
 *const\_iterator сend* ( ) *const*;  
 *reverse\_iterator rbegin* ( ); //установка на первый c конца  
 *const\_reverse\_iterator rсbegin* ( ) *const*;  
 *reverse\_iterator rend* ( ); // — на следующий за последним c конца  
 *const\_reverse\_iterator rсend* ( ) *const*;

Пара соответствующих функций задаёт полуоткрытый интервал, в котором находятся элементы контейнера (начало и конец для параметра цикла: *for* (*auto* *x* = *A.begin*( ); *x* != *A.end*( ); ++*x*) *f*(\**x*); //Обработать всё.

Альтернатива (С++11): *for*(*auto x* : *A*) *f*(*x*); //Здесь *x* — элемент контейнера

Сведения о размере:

*size\_type size* ( ); — количество элементов в контейнере;  
 *size\_type max\_size* ( ); — максимально возможное количество;  
 *bool empty* ( ); — проверка «контейнер пуст».

ИТЕРАТОРЫ: сущность и классификация

Итератор — это класс-указатель, т. е. класс, для которого определены: разыменование прямое \* и косвенное ->, инкремент/декремент ++ и - -, возможно, индексация [ ].

Конкретный набор операций связан с типом (областью применения) итератора:

— входные: только чтение, однопроходный: \* и ++;

ВАРИАНТ: прямые, обратные;

— выходные: однократный проход для записи (только присваивание =);

ВАРИАНТ: итераторы вставки — в начало, в конец, универсальный (используют push\_front( ), push\_back( ), insert( ));

— двунаправленные: инкремент и декремент;

— произвольного доступа: индексация;

— потоковые (входные, выходные).

Применимость к контейнерам — для каждого указывается отдельно!

Template <class T> //Итератор = объект-указатель  
 Class SmartPtr  
 { public:   
 SmartPtr (T \* realptr = nullptr);  
 SmartPtr(const SmartPtr & Rhs);  
 ~SmartPtr( );  
 T& operator \*( ) const; //Разыменование прямое  
 T\* operator->( ) const; //Разыменование косвенное  
 T& operator[ ](int) const; //Прямой доступ (индексирование)  
 private:  
 T\* pointee; //Реальный указатель  
 };

**Последовательные контейнеры — вставки** (применимость)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операция** | **функция** | **vector** | **deque** | **list** |
| Вставка в начало | push\_front | нет | да | да |
| Удаление из начала | pop\_front | нет | да | да |
| Вставка в конец | push\_back | да | да | да |
| Удаление с конца | pop\_back | да | да | да |
| Вставка внутрь | insert | (да) | (да) | да |
| Удаление изнутри | erase | (да) | (да) | да |
| Произвольный доступ | [ ], at | да | да | нет |

Для работы с последовательностью: просмотр, копирование и т. д. все контейнеры эквивалентны, поэтому рекомендуется самый экономичный из них — *vector*. Последний обеспечивает также прямой доступ к данным, что позволяет обрабатывать их в произвольном порядке. Но если предполагается добавлять или удалять данные из начала или середины — вне конкуренции *list*, выполняющий такие операции за константное время. Но произвольного доступа он не обеспечивает, и поиск данных в нём возможен только просмотром с начала. Контейнер *deque* — компромиссный вариант, обеспечивающий как эффективную вставку/удаление в начале, так и прямой доступ к данным.

НЕСТАНДАРТНЫЕ КОНТЕЙНЕРЫ (не подчиняющиеся общим правилам)

— *vector*<*bool*> — упакован по битам и нет доступа к биту через итератор (аналог массива битов);

— *bitset* — аналог машинного слова, любое к-во битов;

— *string* — оболочка для строки, вариант: *wstring* — для 16-битовых символов. Нестандартность — полный набор своих функций-членов, позволяющий обойтись без средств из библиотеки алгоритмов

**Сводка средств для работы с контейнером *string*/*wstring***

*— конструкторы*:

*string*( ); //умолчание (создаётся пустая строка)  
 *string*(*const char* \*); //из строки в стиле Си  
 *string*(*const char* \*, *int n*); //то же, с указанием размера  
 *string*(*string*&); //копирование

*— присваивания*:

*string*& *operator* = (*const string*&);  
 *string*& *operator* = (*const char*\*);  
 *string*& *operator* = (*char*);

Строка в стиле Си преобразуется в *string* по умолчанию;

обратное преобразование — только явно, функцией *c\_str*( ).

*— операции над строками:*

= (присваивание), + (конкатенация), +=, ==, !=, <, ≤, >, ≥, [ ], ≪, ≫.

Размеры строк устанавливаются автоматически так, чтобы результат помещался.

Адрес нулевого элемента строки НЕ совпадает с указателем на строку.  
 *S* != &*S*[0].

Допустимость индекса в операции *S*[ i ] не проверяется. Для индексации с проверкой служит функция *S.at*(*i*), порождающая исключение *out\_of\_range*.

*— получение характеристик строк*:

*size\_type size* ( ) *const*;  
 *size\_type length* ( ) *const*;  
 *size\_type max\_size* ( ) *const*;  
 *size\_type capacity* ( ) *const*;  
 *bool empty* ( ) *const*;

**Функции для строк:**

— присваивание подстроки

*assign* (*const string*& *str*); //=  
 *assign* (*const string*& *str*, *size\_type pos*, *size\_ type n*);  
 *assign* (*const char* \* *s*, *size\_ type n*);

— добавление подстроки

*append* (*const string*& *str*); //то же, что и операция «+»  
 *append* (*const string*& *str*, *size\_type pos*, *size\_ type n*);  
 *append* (*const char* \* *s*, *size\_ type n*);

— вставка подстроки

*insert* (*size\_type pos*1, *const string*& *str*); //вставка целиком  
 *insert* (*size\_type pos*1, *const string*& *str*, *size\_type pos*2, *size\_type n*); //вставка части  
 *insert* (*size\_type pos*1, *const char* \* *s*, *size\_type n*);

— удаление подстроки

*erase* (*size\_type pos* = 0*, size\_type n* = *npos*);   
 //здесь и далее: *npos* — стат.член класса string, содержит максимальное значение *size\_type*.

— очистка строки

*void clear* ( );

— замена подстроки

*replace* (*size\_type pos*1, *size\_type n*1, *const string*& *str*);  
 *replace* (*size\_type pos*1, *size\_type n*1, *const string*& *str*,  
 *size\_type pos*2, *size\_type n*2);  
 *replace* (*size\_type pos*1, *size\_type n*1, *const char* \* *s*, *size\_type n*2);

— обмен строк

*swap* (*string* & *str*);

— выделение подстроки

*string substr* (*size\_type pos* = 0*, size\_type n* = *npos*);

— преобразование в строку в стиле Си

*const char* \* *c\_str* ( ) *const*;  
 *const char* \* *data* ( ) *const*; //не добавляет нуль

— копирование в массив

*size\_type copy* (*char* \* *s*, *size\_type n*, *size\_type pos* = 0 );

//не добавляет нуль

— поиск подстрок

*size\_type find* ( *const string* & *str*, *size\_type pos* = 0) *const*;  
 *size\_type find* ( *char c*, *size\_type pos* = 0) *const*;   
 *size\_type rfind* ( *const string* & *str*, *size\_type pos* = *npos*) *const*;   
 *size\_type rfind* ( *char c*, *size\_type pos* = *npos*) *const*;

# 12. Вспомогательные средства STL

**Пары и кортежи**

*template* <*typename T*1, *typename T*2>  
 *struct pair* { *T*1 *first*; *T*2 *second*; }

Операции над парами:

*pair* <*T*1, *T*2> *p*;  
 *pair* <*T*1, *T*2> *p* (*val*1, *val*2);  
 *pair* <*T*1, *T*2> *p* (*rv*1, *rv*2);  
 *pair* <*T*1, *T*2> *p*(*p*2);  
 *pair* <*T*1, *T*2> *p*(*rv*);

Возможны копирование, перенос, сравнение, обмен;

*make\_pair*(*val*1, *val*2); — создание пары из двух элементов в соответствии с их типами.

Доступ к элементам: *p.first*, *p.second*.

Вариант: *get*<0>(*p*), *get*<1>(*p*) — как для кортежей (*turple*).

**Функциональные объекты**:

— указатели на функцию;

— функторы.

**Функтор** — объект класса с перегруженной операцией ( ).

*Пример*.

class Linear  
 { private:  
 double slope, y0;  
 public:  
 Linear (double s1 = 1, double y = 0) : slope(s1), y0(y) { }  
 double operator( ) (double x) { return y0 + slope \* x; }  
 };

Использование:

Linear f1, f2(2.5, 10.0);  
 double y1 = f1(12.5), y2 = f2(0.4);

*Пример* 2.

// functor.cpp – использование функторов  
 #include <iostream>  
 #include <list>  
#include <iterator>   
 #include <algorithm>  
 template<class T> // класс-функтор определяет operator( )( )  
 class TooBig  
 {  
 private:  
 T cutoff; //Память функтора  
 public:  
 TooBig(const T & t) : cutoff(t) { }  
 bool operator( )(const T & v) { return v > cutoff; }  
 };

void outint(int n) {std::cout << n << " ";}  
int main( )  
 {  
 using std::list;  
 using std::cout;  
 using std::endl;  
 using std::for\_each;  
 using std::remove\_if;

TooBig<int> f100(100); // limit = 100  
 int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};  
 list<int> L1(vals, vals + 10); // конструктор из диапазона   
 list<int> L2(vals, vals + 10);  
// C++11 разрешает инициализировать непосредственно:  
// list<int> L1 = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};  
// list<int> L2 {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};  
 cout << "Оригиналы:\n";  
 for\_each(L1.begin( ), L1.end( ), outint);  
 cout << endl;  
 for\_each(L2.begin(), L2.end(), outint);  
 cout << endl;  
 L1.remove\_if(f100); // исп. поименованный функциональный объект  
 L2.remove\_if(TooBig<int>(200)); // сконструированный функц. объект  
 cout <<"Усечённые:\n";  
 for\_each(L1.begin( ), L1.end( ), outint);  
 cout << endl;  
 for\_each(L2.begin( ), L2.end( ), outint);  
 cout << endl;   
 std::cin.get( );  
 return 0;  
 }

Концепции функторов:

— **генератор** (без аргументов)

— **унарная функция** (один аргумент)

— **бинарная функция** (два аргумента)

— предикат (один аргумент, возврат — bool)

— бинарный предикат (то же, два аргумента)

Примеры из *STL*:

plus, minus, multiplies, divides, modulus, negate — функторы;

equal\_to, not\_equal\_to, greater, less, greater\_equal, less\_equal, logical\_and, logical\_or, logical\_not — бинарные предикаты.

Функтор может применяться как АДАПТЕР функции с неподходящим количеством аргументов.

— **лямбда**

В стандартном C++, например, при использовании алгоритмов стандартной библиотеки C++ *sort* и *find*, часто возникает потребность в определении функций-предикатов рядом с местом, где осуществляется вызов алгоритма. В языке существует только один механизм для этого: возможность определить класс функтора (передача экземпляра класса, определённого внутри функции, в алгоритмы запрещена (*Meyers*, *Effective* *STL*)). Зачастую данный способ является слишком избыточным и многословным и лишь затрудняет чтение кода. Кроме того, стандартные правила C++ для классов, определённых в функциях, не позволяют использовать их в шаблонах и таким образом делают их применение невозможным.

Очевидным решением проблемы явилось разрешение определения лямбда-выражений и лямбда-функций в C++11. Лямбда-функция определяется следующим образом:

[ ](int x, int y) { return x + y; }

Тип возвращаемого значения этой безымянной функции вычисляется как *decltype*(*x*+*y*). Тип возвращаемого значения может быть опущен только в том случае, если лямбда-функция представлена в форме *return expression*. Это ограничивает размер лямбда-функции одним выражением.

Тип возвращаемого значения может быть указан явно, например:

[ ](int x, int y) -> int { int z = x + y; return z; }

В этом примере создаётся временная переменная *z* для хранения промежуточного значения. Как и в обычных функциях, это промежуточное значение не сохраняется между вызовами.

Тип возвращаемого значения может быть полностью опущен, если функция не возвращает значения (то есть тип возвращаемого значения — *void*)

Также возможно использование ссылок на переменные, определённые в той же области видимости, что и лямбда-функция. Набор таких переменных обычно называют замыканием. Замыкания определяются и используются следующим образом:

std::vector<int> someList;  
 int total = 0;  
 std::for\_each(someList.begin(), someList.end(),  
 [&total](int x) { total += x; });  
 std::cout << total;

Здесь вычисляется и выводится сумма всех элементов в списке. Переменная *total* хранится как часть замыкания лямбда-функции. Так как она ссылается на стековую переменную *total*, она может менять её значение.

Переменные замыкания для локальных переменных могут быть также определены без использования символа ссылки &. Это означает, что функция будет копировать их значения, что вынуждает пользователя заявлять о намерении сослаться на локальную переменную или скопировать её.

Для лямбда-функций, гарантированно исполняемых в области их видимости, возможно использование всех стековых переменных без необходимости явных ссылок на них:

std::vector<int> someList;  
 int total = 0;  
 std::for\_each(someList.begin( ), someList.end( ),   
 [&](int x) { total += x; });

Способы внутренней реализации могут различаться, но предполагается, что лямбда-функция сохранит указатель на стек функции, в которой она создана, а не будет работать с отдельными ссылками на переменные стека.

Если вместо [&] используется [=], все используемые переменные будут скопированы, что позволяет использовать лямбда-функцию вне области действия исходных переменных.

Способ передачи по умолчанию можно также дополнить списком отдельных переменных. Например, если необходимо передать большинство переменных по ссылке, а одну по значению, можно использовать следующую конструкцию:

int total = 0;  
 int value = 5;  
 [&, value](int x) { total += (x \* value); }  
 //вызов лямбда-функции с передачей значения

Это вызовет передачу *total* по ссылке, а *value* — по значению.

Если лямбда-функция определена в функции-члене класса, она считается дружественной этому классу. Такие лямбда-функции могут использовать ссылку на объект типа класса и обращаться к его внутренним полям:

[ ](SomeType \*typePtr) { typePtr->SomePrivateMemberFunction(); }

Это будет работать, только если областью создания лямбда-функции является функция-член класса SomeType.

Особым образом реализована работа с указателем *this* на объект, с которым взаимодействует текущая функция-член. Он должен быть явно обозначен в лямбда-функции:

[this]( ) { this->SomePrivateMemberFunction( ); }

Использование формы [&] или [=] лямбда-функции делает *this* доступным автоматически.

Тип лямбда-функций зависит от реализации; имя этого типа доступно только компилятору. Если необходимо передать лямбда-функцию в качестве параметра, она должна быть шаблонного типа, либо сохранена с использованием *std::function*. Ключевое слово *auto* позволяет локально сохранить лямбда-функцию:

auto myLambdaFunc = [this]( ) { this->SomePrivateMemberFunction(); };

**Управление инициализацией объектов** — **класс *initializer\_list***

Пример использования, поясняющий суть дела.

class Set{   
 enum { N = 26};  
 int n = 0;  
 char A[N+1]; //Множество в массиве символов  
public:  
 Set(initializer\_list<char> in)  
 { for(auto p = in.begin( ); p != in.end( ); ++p) A[n++] = p; }  
 //Вариант: *for*(*auto x* : *in*) *A*[*n*++] = *x*;

Использование при объявлении объекта:

Set A{ 'a', 'b', 'd', 'r', 'x' };   
 //Используется конструктор с соответствующим типом аргумента.

Для формального аргумента функции *initializer\_list* в качестве фактического аргумента при вызове используется список инициализации, который функция просматривает как обычный последовательный контейнер.

# 13. Интеллектуальные указатели

Проблема работы с указателями в программах на Си/С++ состоит в том, что указатель живёт своей жизнью независимо от того, на что он указывает. Если значение указателя изменяется или он исчезает по выходе из области своего определения, ресурс, на который он указывал, может стать недоступен для управления и превратиться в неудаляемый мусор. Решить эту проблему пытались ещё в самом первом стандарте языка с помощью указателя *auto\_ptr*, управляющего ресурсом, так, чтобы во всех перечисленных выше случаях изменения или исчезновения указателя ресурс тоже автоматически удалялся. Но сделать из *auto\_ptr* надёжный инструмент не получилось, в первую очередь, из-за отсутствия в языке семантики перемещения.

В стандарте С++17 *auto\_ptr* исключён. Вместо него введено три варианта интеллектуального указателя:

**unique\_ptr**

**shared\_ptr**

**week\_ptr**

Класс *unique\_ptr* — концепция исключительного владения. Гарантирует, что объект и связанные с ним ресурсы принадлежат только одному указателю. Современный аналог *auto\_ptr*.

ПРОБЛЕМА утечки ресурсов:

void f( )  
 { ClassA \* ptr = new ClassA;  
 try {  
 // Работаем с объектом, здесь возможно исключение  
 }  
 catch(…) {  
 delete ptr; throw; //В случае сбоя освободить ресурс  
 }  
 delete ptr; //освобождаем ресурс (сбоя не было)  
 }

АЛЬТЕРНАТИВА (используем интеллектуальный указатель):

#include <memory>  
 void f( )  
 { std∷unique\_ptr<ClassA> ptr(new ClassA); //=make\_unique<ClassA>( );  
 // Работаем с объектом  
 } //*delete* и *catch* не нужны

НЕЛЬЗЯ инициализировать присваиванием:

std∷unique\_ptr<ClassA> ptr = new ClassA;

По умолчанию создаётся пустой указатель.

Можно очистить (освободив ресурс!):

ptr = nullptr;  
 ptr.reset( );

Можно освободить от владения, передав объект обычному указателю:

ClassA \* sp = ptr.release( );

Проверка владения:

if ( ptr ) //→ указатель не пуст!  
 if ( ptr != nullptr ) …  
 if (ptr.get( ) != nullptr) …

Передача владения:

std∷ string \* sp = new string("Hello"); //Обычный указатель  
 std∷unique\_ptr<std∷string> up1(sp); //Так можно  
 std∷unique\_ptr<std∷string> up2(sp);   
 //ОШИБКА: один ресурс на два указателя. Этого следует избегать!

std∷unique\_ptr<std∷string> up2(up1);  
 //Так НЕЛЬЗЯ. Ошибка компиляции  
 std∷unique\_ptr<std∷string> up2(std∷move(up1));  
 //ОК: передача владения через копирование

up2 = up1; //НЕЛЬЗЯ: ошибка компиляции  
 up2 = std∷move(up1);  
//ОК: передача владения через присваивание; к объекту \**up*2 применяется *delete*

Передача владения при вызове функций:

void sink(std∷unique\_ptr<ClassA> up)   
//**Функция-сток**: получение и использование указателя; по выходе из функции ресурс освобождается

{ /\*…\*/ }  
 std∷unique\_ptr<ClassA> up (new ClassA); //Создание ресурса  
 //…  
 sink(std∷move(up)); //Вызов с передачей владения в функцию *sink* std∷unique\_ptr<ClassA> source ( ); //Источник данных через указатель  
 { std∷unique\_ptr<ClassA> up (new ClassA);  
 //Работаем с данными  
 return up; //*std*∷*move*( ) по умолчанию!  
 }  
 void g( )  
 { std∷unique\_ptr<ClassA> p;  
 for ( int i = 0; i < 10; ++i)  
 { p = source( );   
//*p* получает владение возвращённым объектом. Предыдущий — удаляется  
 //…  
 }  
 } //удаляется последний объект, которым владел *p*.

**unique\_ptr как член класса**

ПРИМЕР проблемы утечки ресурсов

class ClassB { //Класс с двумя ресурсами — потенциальная проблема!  
 private:  
 ClassA \* ptr1, \*ptr2;  
 public: //утечка памяти первого объекта в случае сбоя со вторым  
 ClassB( int v1, int v2) : ptr1(new ClassA(v1)), ptr2(new ClassA(v2)) { }  
 ClassB( const ClassB & x) : ptr1(new ClassA(\*x.ptr1)),  
 ptr2(new ClassA(\*x.ptr2)) { }  
 const ClassB & operator = ( const ClassB & x) {  
 \*ptr1 = \*x.ptr1; \*ptr2 = \*x.ptr2;  
 return \*this;  
 }   
 ~ClassB ( ) { delete ptr2; delete ptr1; }  
 }

РЕШЕНИЕ: применить *unique\_ptr*

class ClassB {  
 private:  
 std∷ unique\_ptr<ClassA> ptr1, ptr2;  
 public: //утечка невозможна  
 ClassB( int v1, int v2) : ptr1(new ClassA(v1)), ptr2(new ClassA(v2)) { }  
 ClassB( const ClassB & x) : ptr1(new ClassA(\*x.ptr1)),  
 ptr2(new ClassA(\*x.ptr2)) { }  
 const ClassB & operator = ( const ClassB & x) {  
 \*ptr1 = \*x.ptr1; \*ptr2 = \*x.ptr2; //копируются объекты  
 return \*this; }   
 // ~ClassB ( ) { delete ptr1; delete ptr2; }   
 //деструктор не нужен, по умолчанию всё будет как надо  
 };

**unique\_ptr — работа с массивами**

По умолчанию для удаления применяется *delete*. Язык не различает указатели на скаляр и на массив.

std∷unique\_ptr<std∷string> up(new std∷string[10]); //Ошибка времени выполнения (неправильная инициализация)

ПРАВИЛЬНО:

std∷unique\_ptr<std∷string[ ]> up(new std∷string[10]);  
 //Указатель на массив!

Для доступа к объекту вместо \* и -> следует применять [ ].

std∷cout ≪ "Первая строка" ≪ \*up ≪ std∷endl; //ОШИБКА, так нельзя  
 std∷cout ≪ "Первая строка" ≪ up[ 0 ] ≪ std∷endl; //ОК

Ответственность за корректность индекса — на программисте.

Нельзя инициализировать массив объектами производного типа (как и для обычных массивов).

Можно передавать в указатель собственные средства для удаления объекта.

По умолчанию: *class default\_delete* <*T*> или *class default\_delete* <*T*[ ]>.

template<typename T, typename D = default\_delete <T>> //Первичный шаблон

class unique\_ptr {  
 public:  
 T& operator\* ( ) const;  
 T\* operator -> ( ) const noexcept;  
 // …  
 };  
 template<typename T, typename D>  
 //Частичная специализация для массивов  
 class unique\_ptr <T[ ], D>{  
 public:  
 T& operator [ ] ( ) const;

// …  
 };

*Пример*. Работа с множествами в массивах:

class set1{  
 enum {N = 26}; //Ещё один способ определить константу  
 int n;  
 unique\_ptr<char[ ]> A; //За ресурс теперь отвечает указатель  
 public:  
 set1( ) : n(0), A(new char[N+1]) { }  
 //…  
 };

Класс *set*1 СОДЕРЖИТ ресурс. Деструктор не нужен.

### Ещё об интеллектуальных указателях

Класс *unique\_ptr* — дешёвое и эффективное средство контроля связанного с ним ресурса. Гарантирует, что объект и связанные с ним ресурсы принадлежат только одному указателю и живут ровно столько, сколько живёт сам указатель.

Никаких накладных расходов. Объект *unique\_ptr* — это просто оболочка над обычным указателем, в которой запрещён конструктор копии.

Более сложными являются варианты

*shared\_ptr* — разделяемый указатель;

*week\_ptr* — слабый указатель.

Разделяемый указатель можно копировать, получая группу указателей на один ресурс. Освобождение ресурса происходит, когда исчезает последний из указателей на него. Такое поведение обеспечивается блоком контроля, хранящим счётчик активных указателей на ресурс.

Слабый указатель необходим, чтобы удерживать блок контроля, не захватывая ресурс. Разделяемый и слабый указатели — средство для специфических применений, т. е. там, где это оправдано существом задачи.

# 14. Бинарное отношение на множестве. Графы

Произвольное бинарное отношение на множестве и граф как подходящая модель для этого.

В практических задачах с множествами, как правило, приходится учитывать и обрабатывать их связи. Для строгого математического описания любых связей между элементами двух множеств вводится понятие бинарного отношения.

Бинарное отношение между множествами *A* и *B* называется подмножество *R* их прямого произведения *A* × *B*. В случае *A* = *B* мы говорим просто об отношении R на множестве A.

*a R b* = {<*a*, *b*>: *a* ∈ *A*, *b* ∈ *B* } или *a R b* = {<*a*, *b*>: *a, b* ∈ *A*}

Подходящей моделью для бинарного отношения является граф.

**Граф** — это пара множеств *G* = <*V*, *E*>, где *V* — произвольное множество, а *E* = {{*u*, *v*}: *u*, *v* ∈ *V*, *u* ≠ *v*} — множество пар из элементов множества *V*. Если пара {*u*, *v*} представляет собой множество мощностью 2, граф называется неориентированным, а если это последовательность <*u*, *v*> — ориентированным. Будем обозначать мощность множества вершин |*V*| = *n*, а мощность множества рёбер |*E*| = *m*. Очевидно, что справедливо ограничение *m* = *O*(*n*2).

Вершины {*u*, *v*}, образующие ребро, называются ***смежными***, а само ребро — ***инцидентным*** по отношению к образующим его вершинам, а вершины, в свою очередь, ***инцидентны*** ребру. Количество рёбер, инцидентных вершине, называется её ***степенью***. Вершина, не входящая ни в одно ребро, имеет степень 0 и называется изолированной. В ориентированном графе различают также количество рёбер, входящих в вершину — ***полустепень захода*** — и количество выходящих рёбер — ***полустепень выхода***.

Последовательность попарно смежных вершин образует ***путь*** в графе. ***Длина пути*** равна количеству входящих в него рёбер. Если в последовательности, образующей путь, все вершины различны, путь называется ***элементарным***. Путь, начало и конец которого совпадают, называется ***циклом***. Связный граф без циклов называется ***деревом***, несвязный — ***лесом***.

Если любая пара вершин графа связана путём, граф называется ***связным***. Если для любой пары вершин находятся, по крайней мере, два пути, множества вершин которых не пересекаются, граф — ***двусвязный***.

В связном ориентированном графе (орграф) путь между некоторыми вершинами может быть только в одну сторону. Если же любая пара вершин орграфа связана путями в обе стороны, такой граф называется ***сильно связным***.

Граф с пустым множеством вершин называется ***пустым***, а граф, в котором имеются все возможные рёбра, — ***полным*** графом или ***кликой***.

Граф, множество вершин которого можно разбить на два непустых непересекающихся подмножества таким образом, что концы любого ребра будут принадлежать разным подмножествам, называется ***двудольным***.

Если граф каким-либо из перечисленных свойств не обладает, можно ставить задачу отыскания ***компонент*** — максимальных подграфов, обладающих нужным свойством, например, компонент связности, двусвязности, максимальных клик и т. п.

Графы *G* = < *V*, *E*> и *G*' = < *V*', *E*' > называются ***изоморфными***, если существует биекция *f*: *V*→ *V*' такая, что для любой пары вершин {*u*, *v*} ∈ *E* ⬄ {*f*(*u*), *f*(*v*)} ∈ *E*'.

На свойстве изоморфизма строятся все возможные способы хранения графа в памяти. Перечислим наиболее употребительные из них:

1. Вершины хранятся в массиве, каждый элемент которого — множество рёбер в форме вектора битов. Единичные биты соответствуют рёбрам, инцидентным данной вершине. Альтернатива — массив рёбер, каждое из которых задано вектором инцидентных вершин, которых может быть ровно две. Это — **матрица инциденций** размером *m* × *n*. Это расточительный способ, потому что матрица большей частью состоит из нулей. Но он достаточно компактен и удобен для некоторых задач, например для отыскания вершинного или рёберного покрытия. Способ является естественным для неориентированного графа. Для орграфа следует различать начала и концы рёбер, например, так: «–1» — ребро выходит из вершины, «+1» — ребро входит, «2» — и входит, и выходит (петля).

2. Вершины хранятся в массиве, каждый элемент которого — множество смежных вершин в форме массива (вектора) битов. Это **матрица смежности** размерами *n* × *n*, она может содержать 0 и 1 в любой пропорции. Так, полному графу соответствует единичная матрица. Способ удобен для орграфов. Неориентированные графы хранятся как дважды ориентированные, т. е. их матрица смежности всегда симметрична; она может храниться только верхним треугольником.

3. Вершины хранятся в массиве, каждый элемент которого, кроме самой вершины, множество смежных вершин, обычно в форме списка. Каждый элемент списка содержит поле с номером смежной вершины — индексом массива вершин. Это — **списки смежности**. Но поскольку списки — не обязательная форма, можно применить и векторы; более правильным будет название способа **набор множеств смежности.** Способ удобен, если количество рёбер в графе не очень велико, и требует порядка *O*(*n* + *m*) ячеек памяти.

4. Массив рёбер, каждое из которых задано парой номеров инцидентных вершин, — **массив (вектор) пар**. Требует 2 × *m* ячеек памяти. Способы 3 и 4 также естественны для орграфов. Если требуется неориентированный граф, он хранится как дважды ориентированный: для каждого ребра <*u*, *v*> обязательно хранится и противоположно ориентированное ребро <*v*, *u*>.

5. Разветвляющийся список из вершин, в котором рёбра реализованы посредством указателей. Этот способ применяется главным образом для ациклических орграфов (деревьев), а в общем случае малопригоден без каких-либо дополнений.

Интересной реализацией такого способа является структура Вирта. В её основе — список из вершин. Каждая вершина дополнена списком смежности, каждый элемент которого хранит указатель на смежную вершину — элемент списка вершин. Структура похожа на списки смежности, но не использует массив и номера его элементов для идентификации смежных вершин.

Контрольные вопросы.

1. Граф какого вида больше подходит в качестве модели бинарного отношения на произвольном конечном множестве?
2. Путь в графе — это ...
3. Длина пути между парой вершин в ненагруженном графе — это…
4. Цикл в графе называется элементарным, если…
5. Можно ли для представления графа в памяти использовать машинное слово?
6. Какое машинное представление графа может быть сделано самым компактным?
7. Представление графа в виде множеств смежности считается оптимальным…

# 15. Деревья

***Дерево*** в общем случае — это связный граф без циклов (обычно — не ориентированный), абстрактная структура данных, представляющая собой множество *вершин*, или *узлов*, на которых определены попарные связи — *рёбра*. Будем рассматривать частный случай — корневые упорядоченные деревья. Они отличаются тем, что одна из вершин объявлена ***корнем***. Если в дереве есть корень, его рёбра становятся ориентированными, поскольку у любой последовательности попарно связанных вершин — *пути*, включающем корень, появляется направление от корня или к корню. Для любого узла *v* все вершины дерева на пути в корень, находящиеся ближе к корню, называется *предками*, а дальше от корня — *потомками*. Из пары узлов, связанных ребром, узел ближе к корню — *отец*, дальше от корня — *сын*. У каждого узла может быть несколько сыновей, которые называются *братьями*, или *дочерними узлами,* но только один отец. Корень — это единственный в дереве узел, у которого нет отца. Узлы, у которых нет сыновей, называются *листьями*.

Количество рёбер на пути из корня в узел дерева называется *глубиной* узла, количество рёбер на самом длинном пути в лист — *высотой*. Высота дерева — это высота его корня. Разность между высотой дерева и глубиной узла — это *уровень* узла.

Дерево упорядочено, если упорядочены сыновья любого его узла. Сыновья упорядочены, если их перестановка меняет дерево. Из корневых упорядоченных деревьев наиболее часто используются ***двоичные***, или ***бинарные***. Каждый узел двоичного дерева может иметь не более двух сыновей — левого и правого, причём единственный сын узла — обязательно левый или правый. Более сложный вариант — ***троичное*** дерево, где у каждого узла — не более трёх сыновей: левый, средний, правый — в любой комбинации. Каждый из сыновей может рассматриваться как корень соответствующего поддерева, возможно, пустого.

##### **15.1. Представление дерева в памяти**

Поскольку дерево — это граф, годятся все способы представления в памяти, используемые для графов, с одной оговоркой: в структурах для графов общего вида группы смежных вершин сами по себе не упорядочены.

Но есть способы, специфичные именно для деревьев, в частности, для упорядоченных деревьев.

— пара (тройка) массивов. Вершины нумеруются от 0. В нулевом столбце массива располагается корень. Для каждого узла указываются индексы левого и правого сыновей;

— разветвляющийся список. Каждый узел располагается в свободной памяти и хранит указатели на своих сыновей.

— одномерный массив произвольного содержания. В его нулевом элементе располагается корень. Для каждого *i*-го узла (счёт от 0) сыновья располагаются в позициях 2\*(*i* +1) – 1 и 2\*(*i* +1), а отец — в позиции (*i*/2). Дерево в данном случае «прячется» в алгоритме обработки массива. Можно также получить дерево из массива, взяв в качестве корня его средний по порядку элемент, и построив поддеревья тем же способом из левой и правой половин массива. Здесь важен именно массив, используемый как память прямого доступа.

— альтернативные способы (обсуждаются ниже).

Разветвляющийся список — это естественный и часто применяемый способ для представления дерева в памяти, хотя, как показано ниже, далеко не самы экономный. Следует только заметить, что упомянутый выше способ «пара массивов» — это то же самое. Он получается при перехвате управления памятью при размещении узлов, как это проделывалось в своё время при работе с множествами в списках.

Узлы дерева — объекты, связи между которыми осуществляются через указатели. Для создания дерева достаточно объявить класс «узел дерева», членами которого должны быть указатели на узлы того же типа: «левый» и «правый» (у троичного дерева — «левый», «средний» и «правый»). В узле могут быть и другие данные-члены. Минимально необходимым является тег — метка или номер узла, с помощью которого можно различать узлы в процессе их обработки. Для работы с деревом в целом удобно иметь особый класс «дерево», в котором собираются данные, относящиеся к дереву в целом, и функции-члены для работы с деревом. Чтобы эти функции имели доступ к данным узла, достаточно объявить класс «дерево» дружественным для класса «узел».

//Класс «узел дерева»  
class Node { char d;  //тег узла  
 Node \* lft; // левый сын  
 // Node \* mdl; средний сын (если нужно)  
 Node \* rgt; // правый сын  
public:  
 Node( ) : lft(nullptr), rgt(nullptr) { } // конструктор узла  
 ~Node( ){ if(rgt) delete rgt; // деструктор (уничтожает поддерево)  
 if (lft) delete lft; }  
 Node(const Node &) = delete;  
 Node(Node &&) = delete;  
 Node & operator = (const Node &) = delete;  
 Node & operator = (Node &&) = delete;  
friend class Tree; // дружественный класс «дерево»  
 } ;

// Класс «дерево в целом»  
class Tree   
{ Node \* root; // указатель на корень дерева  
 char num, maxnum; //счётчик тегов и максимальный тег  
 int maxrow, offset; //максимальная глубина, смещение корня  
 char \*\* SCREEN; // память для выдачи на экран  
 void clrscr( ); // очистка рабочей памяти  
 Node\* MakeNode(int depth); // создание поддерева  
 void OutNodes(Node \* v, int r, int c); // выдача поддерева  
 Tree (const Tree &); // фиктивный конструктор копии  
 Tree (Tree &&); //копия с переносом (С++11)  
 Tree operator = (const Tree &) const; // фиктивное присваивание  
 Tree operator = (Tree &&) const; //присваивание с переносом (С++11)  
public:  
 Tree(char num, char maxnum, int maxrow);   
 ~Tree();  
 void MakeTree() // ввод — генерация дерева  
 { root = MakeNode(0); }   
 bool exist( ) { return root != nullptr; } // проверка «дерево не пусто»  
 int DFS( ); // обходы дерева  
 int BFS( );  
 void OutTree( ); // выдача на экран  
};

Кроме данных, в классе *Tree* объявлены скрытые функции-члены: вспомогательные функции, которые не входят в интерфейс и предназначены только для вызова из других функций-членов (эти функции при желании можно объявить членами класса «узел»). Конструкторы копирования и перегрузки присваивания сделаны скрытыми умышленно: попытка создать в программе ситуацию, в которой эти функции могут быть вызваны, приведёт к ошибке на этапе компиляции «нарушение защиты». Современные компиляторы рекомендую более радикальный и надёжный способ — пометить эти функции «=delete».

Конструктор дерева инициализирует параметры разметки и создаёт рабочую память — матрицу символов, необходимую для выдачи изображения дерева на экран.

Tree :: Tree(char nm, char mnm, int mxr):   
 num(nm), maxnum(mnm), maxrow(mxr), offset(40), root(nullptr),   
 SCREEN(new char \* [maxrow])  
 { for(int i = 0; i < maxrow; ++i) SCREEN[ i ] = new char[80]; }

Деструктор дерева уничтожает матрицу символов и запускает деструктор узла для корня.

Tree :: ~Tree( ) { for(int i = 0; i < maxrow; ++i) delete [ ]SCREEN[i];  
 delete [ ] SCREEN; delete root; }

Обратите внимание на то, как создаётся и уничтожается матрица.

##### **15.2. Обходы дерева как рекурсивной структуры данных**

Чтобы обработать каким-либо образом множество узлов дерева, его нужно обойти. Каждый узел дерева является корнем поддерева, а его сыновья — тоже корнями поддеревьев. Поэтому алгоритм обхода, запускаясь для узла, должен обработать информацию в узле и запустить такой же алгоритм для каждого из непустых поддеревьев. Существует три способа сделать это, отличающиеся лишь порядком шагов:

1. Прямой обход:

— обработать узел;

— посетить в прямом порядке каждого сына (левого, среднего, правого).

2. Обратный обход:

— посетить в обратном порядке каждого сына (левого, среднего, правого);

— обработать узел.

3. Внутренний, или симметричный обход:

— посетить во внутреннем порядке левого сына;

— обработать узел;

— посетить во внутреннем порядке правого сына (остальных сыновей).

Минимальная обработка узла может состоять в присвоении соответствующему в нём полю номера в порядке посещения (разметка) или в выдаче номеров на экран, если они уже имеются, или в формировании последовательности из номеров посещённых узлов. Очевидно, что не существует иных способов отличить один порядок обхода узлов от другого.

При разметке дерева в прямом порядке номер любого узла — наименьший, а при обратном — наибольший в соответствующем поддереве, а диапазон использованных номеров равен мощности поддерева. При разметке внутренним способом номер узла больше любого номера в левом поддереве и меньше любого номера в правом.

##### **15.3. Создание дерева**

Для создания дерева в памяти тоже применяется алгоритм обхода. Первым шагом этого алгоритма является проверка необходимости создания узла.

Если ответ положительный, узел создаётся и в нём заполняются информационные поля. В частности, может быть выполнен шаг разметки. Далее заполняются поля указателей на каждого сына: для получения значения указателя алгоритм запускается рекурсивно. Результат — указатель на вновь созданный узел или нуль, если узел не создан.

Проверка необходимости создания узла может быть выполнена тремя способами:

1. Запрос на ввод с клавиатуры. Приглашение ко вводу может содержать какую-либо информацию о месте предполагаемого узла в дереве. Ожидаемый ответ — «да» или «нет» (1 или 0, *Y* или *N*, и т. п.). Вместо ответа «да» можно вводить произвольную информацию (но не разметку!) для размещения в узле, особый ввод, например пустой, может означать «нет».

2. Чтение очередного элемента заранее заготовленной последовательности из массива, линейного списка или файла. Такая последовательность сама по себе тоже является способом размещения дерева в памяти, а алгоритм ввода просто преобразует её в форму разветвляющегося списка.

3. Обращение к датчику случайных чисел с целью генерации дерева. Датчик должен быть управляемым. Простой датчик с равновероятной выдачей 0 или 1 будет создавать пустые или очень маломощные деревья — из 1, 2, 3 узлов, так как вероятность того, что узел будет создан, очень быстро падает с ростом его глубины: для корня она составляет всего 0.5, для сыновей — 0.25 (0.52) и т. д. Нужен датчик, который бы обеспечивал вероятность создания корня близкую к 1 и уменьшал её с ростом глубины узла.

***Пример* такого датчика:** Y = depth < rand() % 6 + 1;

Здесь *depth* — глубина узла: для корня она 0, для произвольного узла — на 1 больше, чем у отца. Очевидно, что для корня *Y* = 1, а для узла на глубине больше 5 — всегда 0.

Функция-член для генерации случайного дерева может выглядеть так:

Node \* Tree :: MakeNode(int depth)  
{ Node \* v = nullptr;  
 int Y = (depth < rand( )%6+1) && (num <= 'z');  
// Вариант: cout ≪ "Node (" ≪ num ≪ ',' ≪ depth ≪ ")1/0: "; cin ≫ Y;  
if (Y) { // создание узла, если Y = 1  
 v = new Node;  
 v->d = num++; // разметка в прямом порядке (= «в глубину»)  
 v->lft = MakeNode(depth+1);  
// v->d = num++; //вариант — во внутреннем  
 v->rgt = MakeNode(depth+1);  
// v->d = num++; // вариант — в обратном  
 }  
 return v;  
}

Эта функция запускается из встраиваемой функции-члена *MakeTree*(), результат её работы присваивается полю *root*.

Вместо генерации случайного значения *Y* можно организовать ввод его с клавиатуры. Соответствующая альтернатива помещена в комментарий.

Функция создаёт дерево прямым обходом по той простой причине, что невозможно создать узел дерева, если не создан его отец. Но вот считать узел «пройдённым» можно когда угодно. Поэтому для разметки узла в алгоритме можно использовать три точки (две из них закомментированы): до обхода поддеревьев, после левого поддерева и перед правым и по окончании обхода поддеревьев. Нужный вариант разметки можно обеспечить, включив инициализацию в соответствующей точке и выключив — в остальных.

Значение глубины узла *depth*, необходимое для датчика, известно при входе в функцию и может быть использовано в любом месте. А вот данные, зависящие от поддеревьев: высота узла, количество листьев, количество потомков и т. п., могут быть известны только тогда, когда оба поддерева уже обработаны, т. е. они доступны только при обратном обходе.

##### **15.4. Вывод изображения дерева на экран монитора**

Чтобы получить наглядное представление о способе разметки дерева, нужно вывести его на экран в виде диаграммы. Можно обойтись для этого текстовым режимом, если принять следующее соглашение. В середине первой строки текста вывести метку корня дерева. В следующей строке — расположить метки левого и правого сыновей в серединах левой и правой половины строки и т. д. Если дерево — троичное, метку среднего сына можно разместить прямо под корнем, и т. д., уменьшая смещение сыновей относительно корня в два раза по отношению к предыдущему ряду. Удобно воспользоваться рекурсивной функцией обхода дерева, которая выдаёт метку узла в некоторой точке экрана (*r*, *c*), а для сыновей добавляет 1 к номеру ряда и смещения к номеру столбца. Смещение удобно вычислять сдвигом некоторой константы *offset* на номер ряда, который совпадает с глубиной узла.

Для выдачи метки в нужную точку экрана можно использовать функцию позиционирования курсора *gotoxy*(*r*, *c*) из библиотеки *conio.h*, предварительно очистив экран функцией *clrscr*(). Но поскольку эти функции есть не во всех оболочках, можно обойтись без них, использовав промежуточную буферную память в виде матрицы символов, как это сделано ниже в примере.

Для того чтобы понять разметку дерева, достаточно вывести узлы 5–6 верхних уровней. Для улучшения читабельности картинки рекомендуется вместо числовых меток использовать буквы латинского алфавита.

Функция-член для вывода изображения дерева на экран может выглядеть так:

void Tree :: OutTree( )  
{ clrscr( );  
 OutNodes(root, 1, offset);  
 for (int i = 0; i < maxrow; ++i)  
 { SCREEN[ i ][ 79 ] = 0;  
 cout << ‘\n’ << SCREEN[ i ];  
 }

cout << ‘\n’;   
}

Она запускает закрытую функцию-член *clrscr*( ), которая готовит матрицу символов, заполняя её точками:

void Tree :: clrscr( )  
{ for(int i = 0; i < maxrow; ++i)  
 memset(SCREEN[ i ], '.', 80);  
}

Далее выполняется закрытая функция *OutNodes*( ), расставляющая метки вершин дерева в матрице символов:

void Tree :: OutNodes(Node \* v, int r, int c)   
{ if (r && c && (c<80)) SCREEN[ r – 1 ][ c – 1 ] = v->d; // вывод метки  
if (r < maxrow) {  
 if (v->lft) OutNodes(v->lft, r + 1, c – (offset >> r)); //левый сын  
 // if (v->mdl) OutNode(v->mdl, r + 1, c); — средний сын (если нужно)  
 if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset >> r)); //правый сын  
 }  
}

Затем матрица символов построчно выводится на экран.

Может получиться следующая картинка:

**.......................................a.......................................  
...................b.......................................h...................  
.........c...................e...................i...................j.........  
..............d.........f.........g...........................................  
...............................................................................  
...............................................................................***Рис. 15.1*. Пример выдачи изображения дерева на экран

Вместо вычисления смещения (*offset* >> *r*) в функции иногда удобнее использовать глобальный массив (*M*[*r*]), в котором смещения можно подобрать более точно, чтобы смежные поддеревья не перекрывались: *int* *M*[ ] {40, 20, 10, 5, 2, 1, 0}. Особенно это важно для троичных деревьев. Запас смещений должен быть достаточен для максимально используемой для вывода глубины узла *maxrow*.

*Примеры*: Двоичные деревья с разными вариантами разметки

**.......................................a.......................................  
...................b.......................................j...................  
.........c...................e...................k...................o.........  
..............d.........f.........i.........l.........m.........p.........s....  
..........................g.........................n.............q.........t..  
.........................h.........................................r...........***Рис. 15.2.* Прямая разметка

**.......................................t.......................................  
...................h.......................................s...................  
.........b...................g...................l...................r.........  
..............a.........e.........f.........i.........k.........o.........q....  
..........................d.........................j.............n.........p..  
.........................c.........................................m...........***Рис. 15.3.* Обратная разметка

**.......................................i.......................................  
...................c.......................................n...................  
.........a...................g...................k...................r.........  
..............b.........d.........h.........j.........m.........o.........s....  
..........................f.........................l.............p.........t..  
.........................e.........................................q...........***Рис. 15.4.* Внутренняя (симметричная) разметка

Вывод на экран с использованием набора смещений

//Данные – ширина экрана и набор смещений  
const int wide = 80, offset[ ]{ 20, 10, 5, 2, 1, 0 };  
void Tree :: OutNodes(Node \* v, int r, int c)   
{ if (r && c && (c < wide)) SCREEN[ r – 1 ][ c – 1 ] = v->d; // вывод метки  
if (r < maxrow) {  
 if (v->lft) OutNodes(v->lft, r + 1, c – (offset[ r ])); //левый сын  
 // if (v->mdl) OutNode(v->mdl, r + 1, c); —) средний сын (если нужно  
 if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset[ r ])); //правый сын  
 }  
}

Смысл: подбором смещений можно отсрочить момент наложения узлов на изображении дерева. Особенно это важно для троичных деревьев.

# 16. Нерекурсивные алгоритмы обхода дерева

Алгоритм обхода дерева можно избавить от рекурсии, явно использовав стек

#include <stack>  
//Прямой обход дерева (он же — «в глубину»)  
int Tree :: DFS( )   
{ int count = 0;  
 stack <Node \*> S; //создание стека указателей на узлы  
 S.push(root); // опускаем корень  
 while (!S.empty( )) //Пока стек не пуст…  
{ Node \* v = S.top( ); S.pop( ); // поднять узел из стека  
 cout << v->d << '\_'; ++count; // выдать тег, счёт узлов  
 if (v->rgt) S.push(v->rgt); // STACK <- (правый сын)  
 if (v->lft) S.push(v->lft); // STACK <- (левый сын)  
 }  
 return count;  
}

ВАЖНО: узлы опускаются в стек в порядке, обратном извлечению, т. е. сперва правый, а потом левый, чтобы сохранить правильный порядок обработки;

— для других шагов порядок произволен, невозможно получить эффект симметричного и обратного обхода перестановкой шагов алгоритма;

— опускание в стек правого сына и последующий его подъём при желании можно исключить.

Замена стека очередью — нерекурсивный обход «в ширину»:

#include <queue>  
int Tree :: BFS( )  
{ int count = 0;   
 queue < Node \* > Q; //создание очереди указателей на узлы  
 Q.put(root); // поместить в очередь корень дерева   
 while (!Q.empty( )) //пока очередь не пуста  
{ Node \* v = Q.front( ); Q.pop( );// взять из очереди,  
 cout << v->d << '\_'; ++count; // выдать тег, счёт узлов   
 if (v->lft) Q.push(v->lft); // Q <- (левый сын)  
 if (v->rgt) Q.push(v->rgt); // Q <- (правый сын)  
 }  
 return count;  
}

Узлы помещаются в очередь в том же порядке, в котором будут извлечены — сперва левый, затем правый.

Результат вывода на экран дерева с ширинной разметкой:

**.......................................a.......................................  
...................b.......................................c...................  
.........d...................e...................f...................g.........  
..............h.........i.........j.........k.........l.........m.........n....  
..........................o.........................p.............q.........r..  
.........................s.........................................t...........***Рис. 16.1.* Изображение дерева с ширинной разметкой

# 17. Альтернативные способы хранения дерева в памяти: вектор пар и вектор битов

Вектор пар в данном случае обобщён до вектора узлов дерева и является альтернативной памятью для них, доступной через перегрузку операций *new* и *delete*. Изменение в коде сводится к определению этих операций в классе *Node*.

Для вектора битов используется обобщённая форма: нуль/не нуль. Использование такой формы хранения дерева в памяти делает ненужными хранение указателей и работу с ними, что даёт существенную экономию. Не нужен и класс «узел». Ненулевые элементы вектора в примере ниже содержат результат разметки, но могут использоваться и иным образом — для произвольной информации, размещаемой в узлах дерева.

Недостатки способа:

1. Вектор битов завязан на интерпретирующий его способ обхода дерева — глубинный или ширинный. Следовательно, набор задач, решаемых на таком дереве, завязан на способ его обхода.
2. Сложность или невозможность решения некоторых задач, завязанных на топологию дерева, например, подсчёт количество узлов с двумя сыновьями.
3. Невозможно менять топологию готового дерева: добавлять или исключать произвольные узлы.

Размер вектора битов определяется мощностью дерева: *n* «единиц» и (*n*+1) для двоичного дерева или (2*n*+1) для троичного нулей. Если дерево — ширинное и почти сбалансировано, все нули собраны в конце вектора, и их можно не хранить, определяя наличие/отсутствие узла по позиции в массиве.

Этот способ — не что иное как упоминавшийся выше способ хранения дерева в массиве произвольного содержания.

//…  
#include <vector>  
#include <string>  
using namespace std;  
  
//Класс «узел дерева»  
class Node { char d; //тег узла  
 Node \* lft; // левый сын  
 Node \* rgt; // правый сын  
 enum {MaxN = 100};  
public:  
 static Node Nodes[MaxN]; //Память для вектора пар (узлов дерева)  
 static int pm; //Мощность дерева = к-во используемой памяти   
 static void \* operator new(size\_t size) { return &Nodes[pm++]; }  
 static void operator delete(void \* pp, size\_t size) { } //Пустой!  
 Node() : lft(nullptr), rgt(nullptr) { } // конструктор узла  
 ~Node(){ if(rgt) delete rgt; // деструктор (уничтожает поддерево)  
 if (lft) delete lft; }  
 static int Outpm( ) { return pm; } //Вывод мощности  
 static void OutNodes( ) { //Вывод содержимого памяти для узлов  
 for (auto i = 0; i < pm; ++i) cout << Nodes[i].d << " ";  
 }  
friend class Tree; // дружественный класс «дерево»  
};  
// Класс «дерево в целом»  
int Node::pm = 0; //Статические данные — инициализация  
Node Node::Nodes[MaxN]; //Альтернативная память под узлы дерева (дерево в векторе пар)  
class Tree   
{ Node \* root; // указатель на корень дерева  
 vector<int>tr; // массив для хранения дерева  
 char num, maxnum; //счётчик тегов и максимальный тег  
 int maxrow, offset; //максимальная глубина, смещение корня  
 int n = 0; //Мощность дерева  
 char \*\* SCREEN; // память для выдачи на экран  
 void clrscr( ); // очистка рабочей памяти  
 Node\* MakeNode(int depth); // создание поддерева  
 void OutNodes(Node \* v, int r, int c); // выдача поддерева  
 Tree (const Tree &) = delete; // фиктивный конструктор копии  
 Tree (Tree &&) = delete; //копия с переносом (С++11)  
 Tree operator = (const Tree &) const = delete; // фиктивное присваивание  
 Tree operator = (Tree &&) const = delete; //присваивание с переносом   
public:  
 Tree(char num, char maxnum, int maxrow);   
 ~Tree( );  
 void MakeTree( ) { root = MakeNode(0); } // ввод — генерация дерева  
 bool exist( ) { return root != nullptr; } // проверка «дерево не пусто»  
 int DFS( ); // нерекурсивные обходы дерева  
 int BFS( );  
 void OutTree(); // выдача на экран  
 void OutTr(int mode = 0) { //Обходы дерева, заданного вектором  
 n = 0;  
 string S[ ]{"\nВектор =", "\nПрямой", "\nВнутренний", "\nОбратный" };  
 cout << S[mode] << " обход TR:<";  
 switch (mode) {  
 case 0: for (auto x : tr) cout << (x ? char(x) : '0') << " "; break;  
 case 1: Tr1( ); break;  
 case 2: Tr2( ); break;  
 case 3: Tr3( ); break;  
 default: cout << " Empty ";  
 }  
 cout << ">\n";  
 }  
 void Tr1( ); //Выдача на экран прямого обхода вектора  
 void Tr2( ); // == симметричного  
 void Tr3( ); // == обратного  
};

void Tree::Tr1( ) { for (auto x : tr) if (x) cout << char(x) << " "; }  
void Tree::Tr2( ) {  
 int pos{ n }; //запоминание текущей позиции  
 if (tr[n++]) {  
 Tr2( );  
 cout << static\_cast<char>(tr[pos]) << " ";  
 Tr2( );  
 }  
}

void Tree::Tr3( ) {  
 int pos{ n }; //запоминание текущей позиции  
 if (tr[n++]) {  
 Tr3( );  
 Tr3( );  
 cout << static\_cast<char>(tr[pos]) << " ";  
 }  
}

Результат прогона программы:

**.......................................a.......................................  
...................b.......................................j...................  
.........c...................e...................k...................o.........  
..............d.........f.........i.........l.........m.........p.........s....  
..........................g.........................n.............q.........t..  
.........................h.........................................r...........  
...............................................................................  
...............................................................................**

**Вектор :<a b c 0 d 0 0 e f 0 g h 0 0 0 i 0 0 j k l 0 0 m n 0 0 0 o p 0 q 0 r 0 0 s 0 t 0 0 >  
Прямой обход TR:<a b c d e f g h i j k l m n o p q r s t >  
Внутренний обход TR:<c d b f h g e i a l k n m j p q r o s t >  
Обратный обход TR:<d c h g f i e b l n m k r q p t s o j a >  
Обход в глубину: a\_b\_c\_d\_e\_f\_g\_h\_i\_j\_k\_l\_m\_n\_o\_p\_q\_r\_s\_t\_ Пройдено узлов = 20  
Обход в ширину: a\_b\_j\_c\_e\_k\_o\_d\_f\_i\_l\_m\_p\_s\_g\_n\_q\_t\_h\_r\_ Пройдено узлов = 20  
pm=20** *Рис. 17.1*. Прогон программы с деревом в векторах пар и битов

#### Альтернативные способы разметки дерева

Поскольку множество мощностью *n* можно в принципе разметить *n*! способами, рассмотренные выше способы составляют очень небольшую их часть. Произвольную (случайную) разметку дерева можно получить, используя генератор перестановок: генерируется мощность *n*, а затем формируется перестановка. Это можно сделать только при хранении дерева в произвольном массиве или использовании этого массива в качестве словаря для разметки дерева, хранящегося другим способом.

**.......................................s.......................................  
...................b.......................................g...................  
.........d...................a...................f...................n.........  
..............c.........f.........j.........k.........l.........m.........p....  
..........................i.........................m.............q.........r..  
.........................e.........................................t...........***Рис. 17.2*. Изображение дерева со случайной разметкой

#### Временная сложность алгоритмов обхода дерева

Все рассмотренные алгоритмы обходят каждый из *n* узлов дерева по одному разу, используя для этого некоторое фиксированное количество шагов. Следовательно, временная сложность каждого обхода — *O*(*n*).

С точки зрения расхода памяти очевидно, что все способы требуют *O*(*n*) элементов памяти. Самый экономный способ — машинное слово, самый расточительный — разветвляющийся список. Но для выбора самого удобного способа в конкретной задаче могут быть и другие основания, которые нужно будет определить самостоятельно.

*Контрольные вопросы*.

1. Какой способ размещения двоичного дерева в памяти является самым универсальным?
2. Какой способ размещения двоичного дерева в памяти является самым компактным?
3. Какой из алгоритмов обхода двоичного дерева с разметкой вершин является оптимальным?
4. Какой обход двоичного дерева рекомендуется применить для определения мощности каждого из образующих его поддеревьев?
5. Какой обход двоичного дерева пригоден для определения расстояния от корня до каждой из вершин?
6. Какой обход двоичного дерева не годится создания его в виде разветвляющегося списка из заданной последовательности битов?
7. Какой алгоритм обхода двоичного дерева пригоден для определения его высоты?
8. Какой алгоритм обхода двоичного дерева не требует использования стека?
9. Какой стек работает быстрее?
10. Сколько вариантов функции *swap* создаёт компилятор при определении её как шаблона, если функция не вызывается?

# 18. Обходы графов

Обход вершин графа может быть выполнен теми же способами, что и обход дерева. Однако следует учитывать, что граф общего вида отличается тем, что он может быть:

— не связен;

— может содержать циклы.

Поэтому нужно создавать дополнительную структуру данных — массив битов и отмечать в нём пройдённые вершины. Если по завершении алгоритма обхода часть осталась не пройдённой, алгоритм перезапускается до тех пор, пока таковых не останется. Количество запусков алгоритма равно количеству компонент связности графа.

class Vertex { int data;   
 Vertex \* next;  
friend class Gr;  
};

class Gr {  
 enum {N = 20};  
 struct Vertex { int data; Vertex \* next;  
 Vertex(int d, Vertex \*v = nullptr) : data(d), next(v) { }  
 } // Вершина (узел)  
 Vertex \* List[N], \* p[N];   
 //Списки смежности и рабочие указатели  
 int Num[N]; //Номера/метки вершин  
 int n = 0, m = 0; //К-во вершин и рёбер  
public:  
 Gr( ) { for (int i=0; i<N; ++i) p[i] = List[i] = nullptr, Num[i]=0; }  
 void GrInit( );  
 void GrOut( );  
 void DFS(int v);  
 void DFS1(int v);  
 void BFS(int v);  
 int traverce(int);  
};

#### Рекурсивный обход

void Gr ∷ DFS(int v)  
{ Num[v] = ++n;  
 for (Vertex \*p = List[v]; p; p = p->next) {  
 int u = p->data; ++m;  
 if(!Num[u]) DFS(u);  
 }  
}

Отличие обхода графа от обхода дерева:

— вместо нескольких шагов с рекурсивным вызовом обработки смежных узлов — цикл по множеству смежных вершин: эффективное представление в памяти — набор множеств смежных в виде списка элементов. Массив в  общем случае потребует памяти под универсум, т. е. *n* элементов, как и матрица смежности. Использование матрицы смежности потребует для каждой вершины n шагов цикла, т. е. всего будет *n*2 шагов. Использование множеств смежности из элементов потребует для этого только *m* или (2\**m*) шагов в сумме по всем циклам, что даёт общую сложность обхода O(*n* + *m*) — что-то среднее между O(*n*) и O(*n*2).

— массив битов необходим, потому что, в отличие от дерева, в вершину графа могут быть несколько путей, а также нужно отличать обработанную часть графа от необработанной;

— массив битов можно использовать в обобщённой форме — как место для хранения результатов разметки вершин графа в процессе обхода. Нуль обозначает не пройдённую вершину, а сама разметка начинается с 1.

Запускающая программа

int Gr ∷ traverce(int mode)  
{ int k = 0;  
 for(int i = 0; i < N; ++i) //Контрольный цикл по всем вершинам  
 if(!Num[i]) { ++k; //Найдена новая; счёт компонент связности  
 switch(mode) { //выбор способа обхода  
 1: DFS(i); break; //в глубину — рекурсивный  
 2: DFS1(i); break; //в глубину — не рекурсивный  
 3: BFS(i); //в ширину  
}  
 m/=2; //Для неорграфа (каждое ребро учитывалось дважды)  
 return k;  
}  
int main( )  
{ Gr A;  
 A.GrInit( );  
 A.GrOut( );  
 cout ≪ “k=” ≪ A.traverce( 1 );  
 }

**Нерекурсивный обход в глубину**

void Gr ∷ DFS1(int v)  
{ stack <int> s;  
 for(int i = 0; i < N; ++i) p[ i ] = List[ i ];  
 s.push(v); Num[ v ] = ++n; //Стартовую – в стек, пройдена   
 while (!s.empty( )) //Пока стек не пуст  
 { int t = s.top( )); s.pop( );  
 bool b;  
 if(p[ t ] == nullptr) b = false;  
 else b = !Num[p[ t ]->data];  
 while(b) { p[ t ] = p[ t ]->next; //Пропуск пройдённых  
 if(p[ t ] == nullptr) b = false;  
 else b = !Num[p[ t ]->data];  
 }  
 if(p[ t ]) {u = p[ t ]->data; s.push(u); //Нашли новую  
 Num[ u ] = ++n; //Обработка  
 }   
 }  
}

Нерекурсивный обход, кроме стека для вершин, требует наличия рабочей копии указателей на проходимые вершины в списках смежности — массива p[v]. Эти указатели инициализируются содержимым массива списков смежности List[v], а затем движутся самостоятельно каждый по своему списку.

#### Замена стека очередью – обход в ширину

void Gr ∷ BFS(int v)  
{ queue <Node \*> q;  
 q.push(v); Num[v] = ++n;   
 while (!q.empty( ))  
 { v = q.front( )); q.pop( );  
 for(Vertex \*p = List[v]; p; p = p->next)  
 { int u = p[v]-> data;   
 if(!Num[u] { q.push(u); Num[u] = ++n; }  
 }  
}

Соображения по вводу/генерации/выводу.

— для задания графа-константы — набор массивов смежности;

— для ввода с клавиатуры (из файла) тоже удобны массивы смежности — далее рекомендуется получение матрицы смежности (устранение дубликатов и симметрирование (для неорграфа), затем — получение списков смежности;

— генерация случайного графа проще всего в форме матрицы смежности;

— вывод результата — в виде массивов или матрицы смежности.

# 19. Стягивающие деревья

*Определение*.

Для произвольного графа G = <V, E> стягивающим деревом называется подграф G' = <V, T>, где T ⊆ E, не имеющий циклов. Если граф G — не связный, G' представляет собой стягивающий лес, т. е. множество деревьев, по одному для каждой компоненты связности графа G.

В результате построения стягивающего дерева множество рёбер исходного графа разбивается на два непересекающихся подмножества:

*T* — множество ветвей;

*E* \ *T* — множество хорд.

Ещё один результат. Если граф G может быть произвольным, то граф G' — всегда ориентированный: вершина, с которой начинается обход графа, становится корнем стягивающего дерева, и у каждого ребра появляется направление — от корня или к корню.

Способ получения множества T:

Обходим граф в глубину или в ширину. Обнаружив новую вершину u, смежную с проходимой v, добавляем в множество T ребро <v, u>.

Пример для рекурсивного обхода:

void Gr ∷ DFS-tr(int v)  
{ Num[v] = ++n;  
 for (Vertex \*p = List[v]; p; p = p->next) {  
 int u = p->data; ++m;  
 if(!Num[u]) { //Вершина *u* — новая  
 { T = T ∪ <v, u>; } //Добавляем в дерево ребро <*v*, *u*>  
 DFS-tr(u);  
 }  
 else //Вершина *u* посещалась, <*v*, *u*> — хорда   
 { }  
 }  
}

К-во вершин в графе |V| = n; к-во рёбер |E| = m;   
к-во компонент связности k.

К-во ветвей стягивающего дерева |T| = n – k; к-во хорд |E\T| = m – n + k.

### Cвойства глубинного и ширинного СД

В неориентированном графе с построенном в нём глубинным деревом любое ребро соединяет предка с потомком на этом дереве. Для ветвей это очевидно: каждая ветвь соединяет отца и сына. Но это справедливо и для всех хорд.

В ориентированном графе ситуация немного сложнее. Хорды могут получаться одного из трёх видов:

— прямая: от предка к потомку (Num[u] > Num[v]; мы нашли альтернативный путь в уже пройдённую вершину);

— обратная: от потомка к предку (Num[u] < Num[v] и вершина u ещё в стеке; мы нашли ребро, замыкающее цикл);

— поперечная: от текущей вершины к пройдённой, не являющейся предком на глубинном дереве (Num[u] < Num[v] и вершина u уже не в стеке; мы нашли ребро, не замыкающее никакого цикла).

Если в графе *G* (независимо от ориентации) построено ширинное стягивающее дерево, кратчайший путь из корня в любую вершину дерева идёт по его ветвям.

*Контрольные вопросы.*

1. Сколько может быть рёбер в неориентированном графе из 100 вершин?
2. Сколько может быть рёбер в стягивающем дереве графа из 100 вершин?
3. Какой алгоритм рекомендуется использовать для определения расстояний от стартовой вершины до всех остальных вершин в неориентированном ненагруженном графе?

# 20. Фундаментальные циклы

*Определение*

Если к стягивающему дереву графа <*V*, *E*> добавить произвольную хорду *e* ∈ *E* \ *T*, то возникший при этом подграф <*V*, *T* ∪ { *e* }> содержит в точности один цикл, который мы обозначим *Ce*.

Множество *C* = { *Ce* : *e* ∈ *E* \ *T* } будем называть фундаментальным множеством циклов графа *G* (относительно стягивающего дерева <*V*, *T*>).

Название «фундаментальный» связано с тем фактом, что из множества *C* можно получить все циклы графа *G*.

Рассмотрим операцию симметрическая разность для произвольных множеств *A* и *B*:

*A* ⊕ *B* = (*A* ∪ *B*) \ (*A* ∩ *B*).

Симметрическая разность множеств *A*1, … *Ak* содержит, независимо от расстановки скобок, в точности те элементы, которые содержатся в нечётном числе множеств *Ai*. Отсюда следует, что можно опустить скобки в выражении

*A1* ⊕ *A2* ⊕ *…* ⊕ *Ak.*

Множество *C* рёбер графа является псевдоциклом, если каждая вершина графа <*V*, *C*> имеет чётную степень. Примеры: пустое множество или произвольный цикл графа.

Легко убедиться, что симметрическая разность любого количества псевдоциклов — это псевдоцикл.

Для графа *G* = <*V*, *E*> со стягивающим деревом <*V*, *T*> произвольный цикл можно представить как симметрическую разность некоторого количества фундаментальных циклов. Другими словами, множество фундаментальных циклов графа представляет собой базис для пространства его циклов.

Уточнение: это пространство является пространством псевдоциклов, т. е. не все его члены являются циклами графа, т.к. он не обязаны быть связными.

### Алгоритм нахождения множества фундаментальных циклов

Алгоритм является обобщением алгоритма построения глубинного стягивающего дерева. Обходим граф, присваивая каждой вершине *v* глубинный номер *NUM*[*v*] и дополнительно помещаем её в стек и удаляем оттуда после использования. Очевидно, что стек в каждый момент будет содержать последовательность вершин на пути от корня стягивающего дерева до текущей вершины. Поэтому, если анализируемое ребро {v, u} замыкает цикл, т. е. ненулевой глубинный номер *NUM*[*u*] < *NUM*[*v*] и *u* не находится под верхушкой стека, то вершина u обязательно находится в стеке, и цикл представлен группой элементов стека от вершины v (верхушки) и до вершины u.

Соответствующую часть стека можно выписать как последовательность вершин, образующих фундаментальный цикл. Но чтобы можно было проделать это, нужно использовать не стандартный адаптер *stack*, а его имитацию на последовательном контейнере *vector*.

**Текст программы отыскания множества фундаментальных циклов**

#include "pch.h"  
#include <iostream>  
#include <stack>  
#include <queue>  
#include <vector>  
#include <list>  
#include <fstream>  
using namespace std;  
  
class Gr {  
 enum {N = 20};  
 vector<list<int>> List;  
 vector<list<int>::iterator> p;  
 vector<int> Num;  
 vector<int> st;  
 int num, k, n, m, kc;  
public:  
 Gr( ) : num(0), k(0), n(0), m(0), kc(0)   
 { Num.resize(N); }  
 void GrInit( );  
 void GrOut( );  
 int FC(int v);  
 int traverce(int);  
 int outm( ) {return m;}  
 int outn( ) {return n; }  
 void OutNum( ) {  
 for(int i = 0; i < n; ++i) cout << char(i + 'a')  
 << "-" << Num[i] << " ";   
 }  
};  
void Gr::GrInit( )  
{ char in[80]={0};  
 vector<vector<int>> M;  
 ifstream infile("in.txt");  
 if(infile.bad( )) exit(1);  
 M.resize(N);  
 for(int i = 0; i < N; ++i) M[i].assign(N,0);  
 int s;  
 do { s = 0;  
 cout << "\n" << char(n+'a') << ":";  
 infile >> in;  
 //infile.get(in, 80);  
 cout << in;  
 for(int i = 0; in[i]; ++i)  
 if(in[i] >= 'a' && in[i] < 'a'+N)   
 M[n][in[i]-'a'] = M[in[i]-'a'][n] = 1, ++s;  
 M[n][n] = 0;  
 if(s) ++n;  
 } while (n < N && s);   
 List.resize(N);  
 for (int i = 0; i < n; ++i){  
 for(int j = 0; j < n; ++j)  
 if(M[i][j]) {  
 List[i].push\_back(j);   
 }  
 }  
}  
void Gr :: GrOut ( )  
{ cout << "\nГраф в виде множеств смежности\n";  
 for(int v = 0; v < n; ++v)  
 if(List[v].size()) {  
 cout << char(v + 'a') << "[ ";  
 for (auto u : List[v])  
 cout << char(u + 'a') << " ";  
 cout << "]\n";  
 }  
}  
void Gr :: DFS(int v)  
{ Num[v] = ++num;  
 for (auto u : List[v]) {  
 ++m;  
 if(!Num[u]) DFS(u);  
 }  
};  
int Gr :: FC( int v)  
{ Num[v] = ++num; st.push\_back(v);  
 for (auto u : List[v]) {  
 ++m;  
 if(!Num[u]) FC(u);  
 else if (Num[u] < Num[v] && u != st[st.size()-2])   
 { ++kc;  
 cout << "\nЦикл " << kc << "= [ ";  
 auto q = st.rbegin(); int t = 0;  
 do { cout << char((t=\*q)+'a') << " "; ++q;  
 } while (t !=u);  
 cout << "]";  
 }  
 }  
 st.pop\_back();  
 return kc;  
};  
int Gr :: traverce(int mode)  
{ switch (mode) {  
 case 1: cout << "\nОбход в глубину\n"; break;  
 case 2: cout << "\nОбход в глубину без рекурсии\n"; break;  
 case 3: cout << "\nОбход в ширину\n"; break;  
 case 4: cout << "\nПолучение фундаментальных циклов\n";  
 break;  
 default: cout << "\nЗадача не определена\n";  
 }  
 for(int i = 0; i < n; ++i) Num[i] = 0;  
 num = 0; k = 0; m = 0; kc = 0;  
 for(int i = 0; i < n; ++i) //Цикл по компонентам связности  
 if(!Num[i]) { ++k;  
 switch (mode) {  
 case 1: DFS(i); break;  
 case 2: DFS1(i); break;  
 case 3: BFS(i); break;  
 case 4: st.clear();  
 cout << "\nНайдено циклов: " << FC(i);  
 break;  
 default: ;  
 }  
 }  
 m /= 2;  
 return k;  
};  
*Пример*. Задан граф в виде набора множеств смежности:

|  |  |
| --- | --- |
|  | a: bcf  b: ace c: bad d: ef e: f f: e |

*Рис. 20.1*. Граф для тестирования алгоритма получения фундаментальных циклов



*Рис. 20.2*. Результат глубинного обхода с построением стягивающего дерева  
и полученные фундаментальные циклы (хорда + ветви дерева)

**Результат прогона программы**

Цикл 1= [ c b a ]  
Цикл 2= [ e d c b ]  
Цикл 3= [ f e d c b a ]  
Цикл 4= [ f e d ]  
Найдено циклов: 4  
Результат обхода: n=6 m=9 k=1  
Порядок обхода: a-1 b-2 c-3 d-4 e-5 f-6

*Контрольные вопросы*

1. Сколько фундаментальных циклов имеется в неориентированном графе из 100 вершин и 1000 рёбер?
2. Какова максимальная длина элементарного пути между произвольной парой вершин в неориентированном графе из 100 вершин и 1000 рёбер?
3. Сколько различных циклов может быть обнаружено в неориентированном графе из 10 вершин и 45 рёбер?
4. Какова временная сложность оптимального алгоритма отыскания фундаментального множества циклов в неориентированном графе из n вершин и m рёбер?

# 21. Двусвязность и сильная связность

Две похожие задачи, основанные на выявлении циклов в графе: двусвязность графов и сильная связность неорграфов.

Хорда (обратная хорда) замыкает цикл, что является источником информации для выделения подграфа с заданным свойством: множество вершин, которые могут принадлежать какому-то циклу.

**Двусвязность**

Вершина *x* неорграфа *G*=<*V*, *E*> — точка сочленения, если найдутся *u*, *v* ∈ *V* => любой путь из *u* в *v* проходит через *x*. Иначе — удаление из графа вершины *x* вместе с инцидентными рёбрами разрывает граф на несколько компонент связности.

Граф без точек сочленения — двусвязный.

Алгоритм отыскания компонент двусвязности основан на обходе графа в глубину с двойной разметкой. Первая разметка — это обычный глубинный номер, назначаемый вершинам в порядке обхода. Вторая разметка — это параметр, представляющий собой наименьшее значение глубинного номера, доступное через хорду. В начале обработки вершины её параметр устанавливается равным глубинному номеру. Затем выполняется цикл по смежным вершинам с рекурсивным вызовом функции обхода для каждой. Перед рекурсивным вызовом в стек опускается ребро — ветвь дерева, а после выполнения вызова сравниваются параметры текущей и обработанной вершины. Если последний меньше, его значение заменяет параметр текущей вершины.

Далее проверяется: номер текущей вершины равен значению параметра обработанной или меньше его. Если это так, текущая вершина является точкой сочленения. Из стека извлекаются рёбра компоненты двусвязности, вплоть до появления ребра <текущая – обработанная>.

Если обрабатываемая вершина — не новая и не отец текущей, то найдена хорда, замыкающая цикл. Хорда тоже опускается в стек, а глубинный номер обрабатываемой вершины заменяет значение параметра текущей (в даль­ней­шем он заменит значение параметра всех вершин компоненты двусвязности в порядке обратного обхода).

**Программа для тестирования алгоритма  
отыскания компонент двусвязности неорграфа**

#include "pch.h"  
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <list>  
#include <stack>  
#include <stdlib.h>  
#include <string>  
using namespace std;  
class Gr {  
 static const int N = 20;  
 vector<list<int>>List; //Множества смежности  
 vector<int> NUM, L;   
 //Глубинный номер и параметр для каждой вершины  
 int num = 0;  
 vector<pair<int, int>>STACK;   
 //Стек рёбер – накопитель для результата  
 // в контейнере *vector* — только для наблюдения  
 int n = 0, m = 0;  
public:  
 Gr() { List.resize(N); }  
 void GrInit();  
 void GrOut();  
 int outm() { return m; }  
 int outn() { return n; }  
 void out\_NUM\_L() {  
 cout << "\nNUM="; for (auto i : NUM) cout << " " << i;  
 cout << "\n L="; for (auto i : L) cout << " " << i;  
 }  
 void DBL(int, int);  
 void traverce() {  
 NUM.resize(n, 0); L.resize(n, 0);  
 for (int i = 0; i < n; ++i) //Цикл компонент связности  
 if (NUM[i] == 0) DBL(i, -1);   
//Здесь (-1) - фиктивный отец для i на глубинном СД  
 }  
};

char nv(int v) { return v + 'a'; }

void Gr::GrInit()   
 //Создание графа вводом множеств смежности из файла  
{  
 string in;  
 vector<vector<int>> M; //Матрица смежности  
 ifstream infile("inDBL.txt");  
 if (infile.bad()) exit(1);  
 M.resize(N);  
 for (int i = 0; i < N; ++i)  
 {  
 M[i].resize(N);  
 for (int j = 0; j < N; ++j) M[i][j] = 0;  
 }  
 int s{ 0 }, v{ 0 };  
 do {  
 s = 0;  
 cout << "\n" << char(v + 'a') << ":";  
 if (!infile.eof())infile >> in;  
 else in[0] = 0;  
 cout << in; //Ввод множества смежности  
 for (auto x : in)   
 //Перенос в матрицу смежности, симметрирование  
 if (x >= 'a' && x < 'a' + N) {   
 //(Недопустимый символ останавливает ввод)  
 int u{ x - 'a' };  
 n = n < u + 1 ? u + 1 : n;  
 M[v][u] = M[u][v] = 1; ++s;  
 }  
 M[v][v] = 0;  
 ++v;  
 } while (n < N && s);  
 List.resize(N);  
 for (int i = 0; i < n; ++i) {   
 //Перевод данных из матрицы в списки смежности  
 for (int j = 0; j < n; ++j)  
 if (M[i][j]) {  
 ++m;  
 List[i].push\_back(j);  
 }  
 }  
 m /= 2;  
}  
void Gr::GrOut()  
{  
 cout << "\nГраф в виде множеств смежности\n";  
 for (int v = 0; v < n; ++v)  
 if (List[v].size()) {  
 cout << nv(v) << "[ ";  
 for (auto u : List[v]) cout << nv(u) << " ";  
 cout << "]\n";  
 }  
}

void Gr::DBL(int v, int p)

//*v* — обрабатываемая вершина, *p* — её отец в глуб.ст.дереве  
{  
 NUM[v] = L[v] = ++num;  
 for (auto u : List[v])  
 {  
 if (!NUM[u])  
 {  
 STACK.push\_back(make\_pair(u, v));   
 //Сохранить в стеке ветвь СД  
 cout << "\nst1:";   
 for (auto i : STACK) cout << " " << nv(i.first)  
 << "-" << nv(i.second);  
 cin.get();  
 DBL(u, v);  
 L[v] = L[u] < L[v] ? L[u] : L[v];  
 //Коррекция L по выходе из рекурсии  
 if (L[u] >= NUM[v])  
 //Найдена точка сочленения - выводим компоненту из стека рёбер  
 {  
 pair<int, int> edge;  
 out\_NUM\_L();  
 cout << "\n<L(" << nv(u)  
 << ") >= NUM(" << nv(v) << ")> [";  
 do {  
 edge = \*STACK.rbegin(); STACK.pop\_back();  
 cout << nv(edge.first)  
 << "-" << nv(edge.second) << "; ";  
 } while (edge != make\_pair(u, v) && !STACK.empty());  
 cout << "]\n";  
 cout << "\nst3:"; for (auto i : STACK) //Остаток стека  
 cout << " " << nv(i.first) << "-" << nv(i.second);   
 cin.get();  
 }  
 }  
 else if ((u != p) && (NUM[u] < NUM[v]))  
 //Найдена хорда (замыкающая цикл)  
 {  
 STACK.push\_back(make\_pair(u, v)); //Сохранить в стеке хорду  
 cout << "\nst2:";  
 for (auto i : STACK)   
 //Контрольный вывод стека с добавленной хордой  
 cout << " " << nv(i.first) << "-" << nv(i.second);   
 cin.get();  
 L[v] = NUM[u] < L[v] ? NUM[u] : L[v];   
 //Глубинный номер копируется в параметр  
 }  
 }  
 cout << "\n< " << nv(v) << "=" << NUM[v] << "/" << L[v] << "> ";  
}

int main()  
{  
 setlocale(0, "Rus");  
 Gr A;  
 cout << "\nDBL test ================= (C)lgn, 25.10.21" <<  
 "\n Введите списки смежности...\n";  
 A.GrInit();  
 A.GrOut();  
 cout << "\n|V|=" << A.outn() << ", |E|=" << A.outm();  
 A.traverce();  
 A.out\_NUM\_L();  
 cout << "\n ===== The End =====\n"; cin.get();  
}



*Рис. 21.1*. Граф для тестирования алгоритма  
отыскания компонент двусвязности и ожидаемый результат

**Результат прогона программы DBL**

DBL test ================= (C)lgn, 25.10.21  
 Введите списки смежности...  
a:bch  
b:ac  
c:abd  
d:cef  
e:dfg  
f:deg  
g:ef  
h:a  
i:.  
Граф в виде множеств смежности

a[ b c h ]  
b[ a c ]  
c[ a b d ]  
d[ c e f ]  
e[ d f g ]  
f[ d e g ]  
g[ e f ]  
h[ a ]  
|V|=8, |E|=10  
st1: b-a  
st1: b-a c-b  
st2: b-a c-b a-c  
st1: b-a c-b a-c d-c  
st1: b-a c-b a-c d-c e-d  
st1: b-a c-b a-c d-c e-d f-e  
st2: b-a c-b a-c d-c e-d f-e d-f  
st1: b-a c-b a-c d-c e-d f-e d-f g-f  
st2: b-a c-b a-c d-c e-d f-e d-f g-f e-g  
< g=7/5>   
< f=6/4>   
< e=5/4>   
NUM= 1 2 3 4 5 6 7 0  
 L= 1 2 1 4 4 4 5 0  
<L(e) >= NUM(d)> [e-g; g-f; d-f; f-e; e-d; ]  
st3: b-a c-b a-c d-c  
< d=4/4>   
NUM= 1 2 3 4 5 6 7 0  
 L= 1 2 1 4 4 4 5 0  
<L(d) >= NUM(c)> [d-c; ]  
st3: b-a c-b a-c  
< c=3/1>   
< b=2/1>   
NUM= 1 2 3 4 5 6 7 0  
 L= 1 1 1 4 4 4 5 0  
<L(b) >= NUM(a)> [a-c; c-b; b-a; ]  
st3:  
st1: h-a  
< h=8/8>   
NUM= 1 2 3 4 5 6 7 8  
 L= 1 1 1 4 4 4 5 8  
<L(h) >= NUM(a)> [h-a; ]  
st3:  
< a=1/1>   
NUM= 1 2 3 4 5 6 7 8  
 L= 1 1 1 4 4 4 5 8  
 ===== The End =====

**Пояснения**:

*st*1 — текущее содержимое стека для результата, добавлена ветвь СД;

*st*2 — то же, добавлена хорда;

*st*3 — то же, извлечена компонента двусвязности;

<*L*(*e*) >= *NUM*(*d*)> [*e-g; g-f; d-f; f-e; e-d*; ] — пара вершин, участвующих в сравнении номера и параметра при принятии решения о выводе результата и сам результат — компонента двусвязности в виде множества рёбер;

< *g*=7/5> — конец цикла обработки указанной вершины: итоговое значение её номера и параметра.

*Контрольные вопросы*

1. Какова временная сложность оптимального алгоритма отыскания фундаментального множества циклов в неориентированном графе из *n* вершин и *m* рёбер?
2. Что является результатом работы алгоритма отыскания компонент двусвязности неорграфа *G* = <*V*, *E*>?
3. Что является результатом работы алгоритма отыскания стягивающего дерева орграфа *G* = <*V*, *E*>?

# 22. Сильная связность в орграфе

*Определение*

Орграф *G*=<*V*, *E*> — сильно связный, если ∀ *u*, *v* ∈ *V* ∃ путь из *u* в *v* и обратно.

Если граф не сильно связен, сильно связными могут быть его компоненты. В этом случае задача сводится к отысканию разбиения множества *V* на набор непересекающихся подмножеств *V*1, *V*2, … *Vk*, образующих подграфы <*Vi*, *Ei*>, где *Ei* = (*Vi* x *Vi*) ∩ *E*.

В сильно связном графе любая вершина должна входить в какой-нибудь цикл. Способ проверки — такой же. как и проверка на двусвязность неорграфа. Строим глубинное дерево и ищем обратные хорды, замыкающие цикл.

Весь путь от корня глубинного дерева до обратной хорды — это цикл, и его можно пометить значением параметра, равного наименьшему глубинному номеру, доступному через обратную хорду.

В орграфе с глубинным деревом, кроме обратных хорд, замыкающих цикл, могут быть прямые и поперечные хорды.

Прямая хорда ведёт от вершины с меньшим глубинным номером в вершину с большим номером, обратная и поперечная — наоборот. Обратная хорда ведёт в вершину, находящуюся в стеке при глубинном обходе. Поперечная — в вершину, которая уже покинула стек, потому что обход для неё завершился и уже перезапущен.

При рекурсивном глубинном обходе обработка вершины начинается с присваивания ей номера и значения параметра, совпадающего с ним. Вершина помещается в стек результата. Затем делается цикл по смежным вершинам и рекурсивный запуск их обработки. По окончании обработки вершины её параметр уменьшается, если величина параметра смежной вершины оказалась меньше.

При обнаружении обратной хорды параметр текущей вершины заменяется глубинным номером смежной. Обратная хорда обнаруживается по номеру смежной вершины, который должен быть меньше номера текущей, и смежная должна быть ещё в стеке. Для последней проверки каждая вершина в начале обработки помечается «в стеке», а в конце обработки отметка снимается.

Если по окончании обработки номер вершины оказался равен её параметру, из стека с результатом выписывается множество вершин компоненты сильной связности — от верхушки до номера последней смежной вершины, который обязательно должен там быть, или до исчерпания стека.

**Программа для отыскания компонент сильной связности орграфа**

#include <iostream>  
#include <fstream>  
#include <vector>  
#include <list>  
#include <stack>  
#include <string>  
using namespace std;

class Gr {  
 static const int N = 20;  
 vector<list<int>>List; //Набор множеств смежности  
 vector<int> NUM, L; //Глубинный номер и параметр   
 int num = 0;

vector<int>STACK; //Стек для результата  
 vector<int>InStack; //Пометки «в стеке»  
 int n = 0, m = 0;  
public:  
 void GrInit();  
 void GrOut();  
 int outm() { return m; }  
 int outn() { return n; }  
 void out\_NUM\_L() {  
 cout << "\nNUM="; for (auto i : NUM) cout << " " << i;  
 cout << "\n L="; for (auto i : L) cout << " " << i;  
 }  
 void FND(int);  
 void traverce() { //Цикл компонент сильной связности  
 NUM.resize(n, 0); L.resize(n, 0);  
 InStack.resize(n, 0);  
 for (int i = 0; i < n; i++) if (!NUM[i]) FND(i);  
 }  
};

char nv(int v) { return v + 'a'; }

void Gr::GrInit()  
{  
 string in;  
 vector<vector<int>> M;  
 ifstream infile("in.txt");  
 if (infile.bad()) {  
 cout << "\nФайл in.txt недоступен"; exit(1);  
 }  
 M.resize(N);  
 for (int i = 0; i < N; ++i)  
 {  
 M[i].resize(N);  
 for (int j = 0; j < N; ++j) M[i][j] = 0;  
 }  
 int s(0), v(0);  
 do {  
 s = 0;  
 cout << "\n" << nv(v) << ":";  
 if (!infile.eof())infile >> in;  
 else in[0] = 0;  
 cout << in;  
 for (int i = 0; in[i]; ++i)  
 if (in[i] >= 'a' && in[i] < 'a' + N) {  
 int u(in[i] - 'a');  
 n = n < u + 1 ? u + 1 : n;  
 M[v][u]   
 // = M[u][v] — симметрирование для неорграфа  
 = 1; ++s;   
 }  
 M[v][v] = 0; //Очистка диагонали – для неорграфа  
 ++v;  
 } while (n < N && s);  
 List.resize(n);  
 for (int i = 0; i < n; ++i) {  
 // List[i].clear();  
 for (int j = 0; j < n; ++j)  
 if (M[i][j]) {  
 ++m;  
 List[i].push\_back(j);  
 }  
 }  
}

void Gr::GrOut()  
{  
 cout << "\nГраф в виде множеств смежности\n";  
 for (int v = 0; v < n; ++v)  
 if (List[v].size()) {  
 cout << char(v + 'a') << "[ ";  
 for (auto u : List[v])  
 cout << nv(u) << " ";  
 cout << "]\n";  
 }  
}  
int main()  
{  
 setlocale(0, "Rus");  
 Gr A;  
 cout << "\nFND test ====================== (C)lgn, 10.10.03;03.11.18" <<  
 "\n Введите списки смежности...\n";  
 A.GrInit();  
 A.GrOut();  
 cout << "\n|V|=" << A.outn() << ", |E|=" << A.outm(); A.traverce();  
 A.out\_NUM\_L();  
 cout << "\n ===== The End =====\n"; cin.get();  
}

void Gr::FND(int v)  
{  
 NUM[v] = L[v] = ++num;  
 STACK.push\_back(v); InStack[v] = true;  
 cout << "\nst1:"; for (auto x : STACK) cout << nv(x); cin.get();  
 for (auto u : List[v])  
 {  
 if (!NUM[u])  
 {  
 FND(u);  
 L[v] = L[u] < L[v] ? L[u] : L[v];  
 }  
 else if (NUM[u] < NUM[v])  
 {  
// for (auto x : STACK) if (x == u) f = 1; прямая проверка

if (InStack[u])//Замена глубинным номером  
 L[v] = NUM[u] < L[v] ? NUM[u] : L[v];  
 }  
 }  
 if (L[v] == NUM[v])  
 //в результате обхода подграфа параметр не изменён!  
 { //Выдача результата  
 int x = 0;  
 out\_NUM\_L();  
 cout << "\n<" << nv(v) << ">[";  
 do {  
 x = \*STACK.rbegin();  
 STACK.pop\_back(); InStack[x] = false;  
 cout << nv(x) << " ";  
 } while ((x != v) && (STACK.size()));  
 cout << "]";  
 cout << "\nst2:";   
 for (auto x : STACK) cout << nv(x) << " ";  
 cin.get();  
 }  
 else cout << " <" << nv(v)   
 << "=" << NUM[v] << "/" << L[v] << ">";  
}

*Пример* 22.1. Тест для алгоритма отыскания сильно связных компонент.

|  |  |
| --- | --- |
|  |  |
| *Рис*. *22.1*.  Выделены ветви стягивающего дерева.  Дана классификация хорд.  Указаны номер и параметр каждой вершины. | Файл in.txt bcd ad d b cf dg ef . |

Результат прогона программы FND

Граф в виде множеств смежности  
a[ b c d ]  
b[ a d ]  
c[ d ]  
d[ b ]  
e[ c f ]  
f[ d g ]  
g[ e f ]

|V|=7, |E|=13  
st1:a  
st1:ab  
st1:abd <d=3/2> <b=2/1>  
st1:abdc  
 <c=4/3>  
NUM= 1 2 4 3 0 0 0  
 L= 1 1 3 2 0 0 0  
<a>[c d b a ]  
st2:  
st1:e  
st1:ef  
st1:efg  
 <g=7/5> <f=6/5>  
NUM= 1 2 4 3 5 6 7  
 L= 1 1 3 2 5 5 5  
<e>[g f e ]  
st2:  
NUM= 1 2 4 3 5 6 7  
 L= 1 1 3 2 5 5 5  
 ===== The End =====

**Временная сложность рассмотренных алгоритмов для графов**

Во всех случаях основой является обход графа в глубину, осуществляющий обработку каждой вершины и каждого ребра графа. Обход в глубину графа, представленного набором множеств смежности — номеров смежных вершин — имеет временную сложность O(*n* + *m*).

Дополнительные действия — формирование и выдача результата.

Это все рёбра графа в сумме для компонент двусвязности — O(*m*), все вершины для компонент сильной связности — O(*n*). Таким образом, общая ожидаемая сложность для этих алгоритмов O(*n*+*m*). Для алгоритма отыскания фундаментальных циклов количество циклов равно количеству хорд, каждый цикл состоит из порядка *n* рёбер, что даёт итоговую оценку сложности алгоритма O((*m*–*n*+1) \* *n*) = O(*m*\**n*).

# 23. Эйлеров цикл

Элементарный цикл по всем рёбрам графа.

Условие существования: все вершины графа должны иметь чётную степень (связный псевдоцикл). В ориентированном графе эйлеров цикл существует, если граф связный и для любой его вершины полустепень выхода равна полустепени захода.

Если в графе нечётную степень имеют только две вершины, они будут началом и концом эйлерова пути (полуэйлеров граф). В таком графе перед обработкой можно временно соединить вершины нечётной степени ребром, может быть, через дополнительную вершину, если ребро уже есть, и применить алгоритм отыскания эйлерова цикла.

Алгоритм начинает обход графа с произвольной вершины *v.* Из множества смежности берётся первый доступный элемент *v* и опускается в рабочий стек, а ребро <*v,* *u*> из графа исключается. Далее *v* = *u*, и процесс повторяется для новой вершины. Если множество смежных для *v* пусто, верхушка рабочего стека извлекается и опускается в стек результата. Новой вершиной *v* будет верхушка рабочего стека. Алгоритм заканчивает работу, если рабочий стек пуст. Результат — в стеке результата — содержит эйлеров путь (в обратном порядке). Поэтому вместо стека для результата можно использовать последовательный контейнер, допускающий вставку в начало. Количество повторений цикла равно количеству рёбер, и внутренняя часть цикла выполняется за постоянное время, следовательно, ожидаемая временная сложность алгоритма O(*m*). Это значит, что алгоритм является эффективным.

**Программа отыскания эйлерова цикла**

class Gr {  
 static const int N = 20;  
 vector<list<int>>List;  
 stack<int>STACK;  
 list<int>result;  
 int n = 0, m = 0;  
public:  
 void GrInit();  
 void GrOut();  
 int outm() { return m; }  
 int outn() { return n; }  
 void CE(int);  
};

void Gr::CE(int v)  
{  
 STACK.push(v);  
 while (!STACK.empty())  
 {  
 int v = STACK.top();  
 if (!List[v].empty())  
 {  
 int u = List[v].front();  
 STACK.push(u);  
 List[v].pop\_front();  
 List[u].remove(v); //Добавка для неорграфа!  
 }  
 else { STACK.pop(); result.push\_front(v); }  
 }  
 cout << "\n Результат:";  
 for (auto x : result) cout << nv(x) << " ";  
 cout << endl;  
}

*Пример* 23.1. Тест алгоритма отыскания эйлерова цикла.

|  |  |
| --- | --- |
| *Рис*. *23.1*. | Файл in.txt bcdh ac abdf acef dfgh cdeg ef ae . |

Результат прогона программы CE

Граф в виде множеств смежности  
a[ b c d h ]  
b[ a c ]  
c[ a b d f ]  
d[ a c e f ]  
e[ d f g h ]  
f[ c d e g ]  
g[ e f ]  
h[ a e ]  
|V|=8, |E|=13  
 Результат: a b c a d c f d e f g e h a

Поскольку в наборе множеств смежности неорграф хранится как дважды ориентированный, удаление ребра означает: надо удалить вершину *u* из множества смежных для *v* и удалить вершину *v* из множества смежных для *u*.

Если первое выполняется за константное время, т. к. в качестве *u* взята первая доступная вершина, то для удаления *v* в неорграфе надо просмотреть всё множество смежных для *u* — за время O(*n*). Чтобы само удаление выполнялось за константное время, множества смежности должны быть представлены списками.

В алгоритме для неорграфов можно использовать дополнительную память — матрицу смежности. Удаление ребра отмечается в ней заменой 1 на 0. Из списков смежности ребро <*v*, *u*> удаляется немедленно, а удаление ребра <*u*, *v*> откладывается до момента, когда оно действительно понадобится. Для каждой очередной вершины *u* наличие ребра <*v*, *u*> проверяется. Если ребро <*v,* *u*> помечено как удалённое, вершина *u* удаляется из списка смежности и берётся следующая по порядку. Таким образом, удаление ребра будет выполняться за константное время, и общая сложность алгоритма отыскания эйлерова цикла останется O(*m*).

# 24. Гамильтонов цикл

Элементарный цикл, содержащий все вершины графа.

В отличие от эйлерова цикла, для этой задача не существует полиномиального критерия наличия решения. Единственный способ проверить граф на наличие гамильтонова цикла — попытаться построить его.

Это можно сделать, модифицировав алгоритм обхода графа способом «в глубину». В процессе обхода в глубину в стеке *X* формируется элементарный путь от стартовой вершины до текущей. Если путь собрал все вершины графа и существует ребро от конечной вершины к начальной, то гамильтонов цикл найден. Если указанного ребра не существует, найден гамильтонов путь, не являющийся циклом.

Чтобы путь *X* был элементарным, в процессе обхода вершины помечаются как «использованные». Алгоритм обхода в глубину найдёт единственный путь, который не обязательно будет решением. Нужно, чтобы алгоритм проверил все существующие пути от стартовой вершины. Для этого достаточно после рекурсивной обработки вершины объявить её вновь «не использованной». Такая добавка сделает алгоритм переборным, и он проверит все пути, которые могут быть построены от стартовой вершины, и найдёт гамильтонов путь. если он существует.

**Алгоритм отыскания гамильтонова цикла**

class Gr {  
 static const int N = 20;  
 vector<list<int>>List;  
 vector<int> NEW, X;  
 int num = 0;  
 int n = 0, m = 0;  
public:  
 void GrInit();  
 void GrOut();  
 int outm() { return m; }  
 int outn() { return n; }  
 void GAM(int);  
};

char nv(int v) { return v + 'a'; }  
void Gr::GrInit()  
{  
 string in;  
 vector<vector<int>> M;  
 ifstream infile("in.txt");  
 if (infile.bad()) exit(1);  
 M.resize(N);  
 for (int i = 0; i < N; ++i)  
 {  
 M[i].resize(N);  
 for (int j = 0; j < N; ++j) M[i][j] = 0;  
 }  
 int s(0), v(0);  
 do {  
 s = 0;  
 cout << "\n" << char(v + 'a') << ":";  
 if (!infile.eof())infile >> in;  
 else in.clear();  
 cout << in;  
 for (auto x : in)  
 if (x >= 'a' && x < 'a' + N) {  
 int u(x - 'a');  
 n = n < u + 1 ? u + 1 : n;  
 M[v][u] = M[u][v] = 1; ++s;  
 }  
 M[v][v] = 0;  
 ++v;  
 } while (n < N && s);  
 List.resize(n);  
 NEW.resize(n, 1); X.resize(n); NEW[0] = 0;  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j)  
 if (M[i][j]) {  
 ++m;  
 List[i].push\_back(j);  
 }  
 }  
 m /= 2;  
}

void Gr::GrOut()  
{  
 cout << "\nГраф в виде множеств смежности\n";  
 for (int v = 0; v < n; ++v)  
 if (List[v].size()) {  
 cout << nv(v) << "[ ";  
 for (auto u : List[v])  
 cout << nv(u) << " ";  
 cout << "]\n";  
 }  
}

void Gr::GAM(int k)  
{  
 if (k == n) //получено решение  
 {  
 cout << "<";  
 for (auto x : X) cout << " " << nv(x);  
 cout << ">\n"; cin.get();  
 }  
 else  
 for (auto u : List[X[k - 1]]) //Цикл по смежным  
 if (NEW[u]) //Найдена новая вершина – используем  
 {  
 NEW[u] = 0; //Вершина *u* «использована»  
 X[k] = u;  
 GAM(k + 1);  
 NEW[u] = 1; //Вершина u вновь свободна  
 //Добавка, делающая алгоритм переборным  
 }  
}

int main()  
{  
 setlocale(0, "Rus");  
 Gr A;  
 cout << "\nGAM test ========== (C)lgn, 16.10.03;01.11.21"   
 << "\n Введите списки смежности...\n";  
 A.GrInit();  
 A.GrOut();  
 cout << "\n|V|=" << A.outn() << ", |E|=" << A.outm() << endl;  
 A.GAM(1);  
 cout << "\n ===== The End =====\n"; cin.get();  
}

*Пример* 24.1. Тестирование программы отыскания гамильтонова цикла

|  |  |
| --- | --- |
| *Рис. 24.1*. | Файл in.txt  bch ac abd cef dfg deg efh ag . |

Результат прогона программы GAM. Найдено 4 решения.

Граф в виде множеств смежности

a[ b c h ]  
b[ a c ]  
c[ a b d ]  
d[ c e f ]  
e[ d f g ]  
f[ d e g ]  
g[ e f h ]  
h[ a g ]

|V|=8, |E|=11  
< a b c d e f g h>  
< a b c d f e g h>  
< a h g e f d c b>  
< a h g f e d c b>  
 ===== The End =====

*Контрольные вопросы*

1. Какова временная сложность оптимального алгоритма отыскания компонент двусвязности в неориентированном графе из *n* вершин и *m* рёбер?
2. Какова временная сложность оптимального алгоритма отыскания сильно связных компонент в ориентированном графе из n вершин и m рёбер?
3. Существует ли эйлеров путь в неориентированном графе, являющемся псевдоциклом?
4. Какова временная сложность оптимального алгоритма отыскания эйлерова пути в неориентированном графе из n вершин и m рёбер?
5. Существует ли эйлеров путь в связном ориентированном графе?
6. Какова временная сложность оптимального алгоритма отыскания гамильтонова пути в неориентированном графе из n вершин и m рёбер?

# 25. Переборные алгоритмы на графах

Для очень многих задач не найдено эффективного алгоритма, т. е. алгоритма полиномиальной сложности.

Классические примеры: подбор пароля и взлом шифра.

Переборной задачей является и тестирование любой более или менее сложной программы как доказательство её корректной работы при любых возможных исходных данных.

Большинство задач для графов тоже являются переборными.

Вот примеры таких задач:

— хроматическое число: количество цветов для раскраски вершин графа с выполнением условия, чтобы концы любого ребра были разных цветов;

— изоморфизм двух графов и изоморфизм подграфу;

— максимальная клика (полный подграф) — частный случай проверки графа на планарность, т. е. возможность его изображения на плоскости без пересечений рёбер;

— минимальное множество вершин, разрезающих циклы орграфа, т. е. вершины, удаление которых делает граф ациклическим;

— максимальное паросочетание — множество рёбер графа, не имеющих общих вершин;

— минимальное вершинное покрытие: множество вершин, содержащих конец любого ребра;

— максимальное независимое множество вершин — множество, в котором никакая пара не связана ребром.

Для решения задачи, для которой не известен эффективный алгоритм, можно применить следующие подходы:

1. Задача, решением которой является некоторая перестановка элементов полного множества. Примеры: проверка графов на изоморфизм, отыскание гамильтонова цикла и т. п. Решение такой задачи может быть сведено к генерации перестановок вершин графа и проверке каждой перестановки, не является ли она решением.

2. Задача, решением которой является подмножество. Здесь можно использовать генератор всех подмножеств.

Эти способы имеют по крайней мере экспоненциальную сложность.

ВАРИАНТ: недетерминированный алгоритм. Генерировать и проверять случайные решения. Остановиться, когда решение будет найдено или будет исчерпано отведенное для этого время.

СТРАТЕГИЯ: используя всю имеющуюся информацию о задаче и результаты испытаний, попытаться отбрасывать заведомо негодные решения до их проверки.

В литературе обсуждаются подходы под названием «метод ветвей и границ», «альфа-бета-отсечение» и т. п.

Один из таких подходов — «перебор с возвратом».

Для его применения требуется, чтобы:

— решение можно было получать в виде расширяемого вектора *X* из *k* элементов, добавляя их по одному к исходному пустому вектору;

— для каждой позиции *k* задано конечное множество допустимых значений *x* ∈ *Ak*и простая функция ρ (*X*, *x*), возвращающая значение *false*, если добавление *x* к вектору заведомо не ведёт к решению, и *true*, если *x* подходит как кандидат. В последнем случае *x* добавляется к вектору *X* в позицию *k*, и производится попытка дальнейшего расширения вектора позицией (*k*+1). Если все подходящие значения *x* проверены, происходит возврат к подбору в позиции (*k*–1).

Именно такую стратегию реализует рассмотренный выше алгоритм отыскания гамильтонова цикла.

В расширяемом векторе *X* формируется путь. Ищется вершина *u*, которую можно поместить в позицию *k*. В качестве множества допустимых значений *Ak* используется множество вершин, смежных для *Xk*–1, а для отбора кандидатов — флажок *NEW*[*u*].

Если решение существует, алгоритм перебора с возвратом обязательно его найдёт. Если решений несколько, будут найдены все. Для полного графа алгоритм сгенерирует все перестановки вершин, каждая из которых будет гамильтоновым путём.

Алгоритм работает значительно быстрее полного перебора за счёт отбрасывания заведомо негодных альтернатив, но его временная сложность всё равно остаётся экспоненциальной, в том числе и при останове после обнаружения первого решения.

Поэтому на практике часто используются приближённые алгоритмы полиномиальной сложности, которые теоретически не могут дать точного решения. Перебор с возвратом позволяет проверить их надёжность.

Возможна и обратная ситуация: перебор с возвратом — простая альтернатива сложному полиномиальному алгоритму в случае, если размерность задачи не очень велика.

# 26. Алгоритмы на графах с нагруженными вершинами. Сортировка

Граф *G*=<*V*, *E*> называется нагруженным, если к его вершинам и/или рёбрам приписаны произвольные элементы некоторого множества — веса, относительно которых формулируется задача. В качестве весов как правило используются вещественные числа.

Рассматриваются неорграфы с весами, приписанными к вершинам, и задачи относительно этих весов.

Задача сортировки: дана произвольная последовательность *S* мощностью *n*. Требуется найти такую перестановку *S*\* = *P*(*S*), что ∀ *i* ∈ [*1*, *n*) *S*\**i*-1 ≤ *Si*. Это возможно только в том случае, если на множестве *U*, *S* ⊆ *U* задано отношение *R* — отношение полного порядка (обозначаемое знаком «≤»).

Это означает, что отношение *R* должно обладать следующими свойствами:

— рефлексивность *aRa*;

— транзитивность *aRb* ˄ *bRc* => *aRc*;

— антисимметричность *aRb* ˄ *bRa* => *a* = *b*

и дополнительным свойством ∀ *a*, *b aRb* ˅ *bRa*

Различают случаи сортировки с учётом или без учёта структуры элементов сортируемой последовательности *S*. Если элементы *S* представляют собой «атомы», т. е. могут сравниваться только целиком, любой алгоритм сортировки сводится к последовательности попарных сравнений с целью выбора нужной перестановки. Для случая *S* = <*a*, *b*, *c*> дерево выбора будет выглядеть так:



*Рис*. *26.1*. Дерево выбора для сортировки трёх элементов

В общем случае дерево выбора для последовательности из *n* элементов должно иметь *n*! листьев, соответствующих каждой из возможных перестановок исходной последовательности. Количество шагов, необходимых для получения результата сортировки, равно высоте этого дерева, т. е. log(*n*!). Можно заметить, что *n*! ≥ (*n*/2)*n*/2, следовательно, log(*n*!) ≥ (*n*/2)log(*n*/2) = (*n*/2) *log* *n* – *n*/2. Время сортировки, основанной на попарном сравнении элементов последовательности, не может быть меньше Ω (*n* log *n*).

Популярные алгоритмы сортировки пузырьком, выбором, вставками и т. п. имеют квадратичную сложность, следовательно, не могут считаться эффективными.

## 26.1. Дерево сортировки и сортировка деревом

Двоичное дерево, узлы которого нагружены таким образом, что вес каждого узла не меньше веса любого узла в его поддереве – дерево сортировки (сортдерево). Другое название — пирамида.

Последовательность весов на любом пути от предка к потомку линейно упорядочена, максимум в корне.



*Рис*. *26.2*. Сортдерево из 13 элементов

Удалим из дерева корень и заменим его крайним справа листом.



*Рис*. *26.3.* Первый шаг сортировки — замена корня листом.

Форма сортдерева нарушена. Обменяем вес в корне с большим из весов сыновей. Если форма не восстановилась, проделаем это и с изменённым весом сына — вплоть до листа.



*Рис. 26.4*. Форма сортдерева восстановлена.

Повторяем то же действие с новым корнем: откладываем его вес и заменяем весом правого листа.



*Рис*. *26.5*. Второй шаг сортировки.



*Рис*. *26.6*. Восстановление после второго шага.



*Рис*. *26.7.* Третий шаг сортировки.



*Рис*. *26.8.* Восстановление после третьего шага.



*Рис. 26.9*. Четвёртый шаг.



*Рис*. *26.10*. Восстановление.



*Рис*. *26.11*. Сделано 13 шагов. Дерево пусто.   
Последовательность весов упорядочена.

Подходящая структура данных для хранения дерева с нагруженными вершинами – линейный массив *S* = <*s*1, …, *s+*>, где *s*1 – вес в корне дерева, а для узла с весом *si* веса сыновей соответственно *s*2*i* и *s*2*i*+1. Из дерева на рис. 26.2 массив получается обходом в ширину.

*S*=< 61 54 57 22 32 33 6 19 1,5 7,3 8 14 31>

Дерево в массиве S является сортдеревом, и его первый элемент – наибольший по величине. Поменяем местами первый и последний элементы (в дереве это соответствует обмену весами корня с крайним справа листом на самом нижнем уровне).

*S*=< **31** 54 57 22 32 33 6 19 1,5 7,3 8 14 **61**>

Для корня тем самым нарушится свойство сортдерева. Восстановим это свойство, обменяв местами вес корня и больший из весов его сыновей.

*S*=< **31** 54 **57** 22 32 33 6 19 1,5 7,3 8 14 **61**>

Для корня свойство восстановится, но, оно нарушится для сына, участвующего в обмене.

*S*=< 57 54 **31** 22 32 **33** 6 19 1,5 7,3 8 14 **61**>

В последнем случае выполняем рекурсивно то же действие для этого сына как корня некоторого поддерева.

S=< 57 54 33 22 32 **31** 6 19 1,5 7,3 8 14 **61**>

Очевидно, что в процессе наведения порядка мы либо остановимся где-нибудь, либо дойдём до листа, т. е. выполним не более чем log *n* обменов. В результате получим наибольший элемент в конце массива и дерево сортировки в остальных (*n*–1) элементах.

Повторяем для нового корня.

S=< **14** **54** 33 22 32 31 6 19 1,5 7,3 8 **57 61**>

S=< 54 **14** 33 22 **32** 31 6 19 1,5 7,3 8 **57 61**>

S=< 54 32 33 22 **14** 31 6 19 1,5 7,3 8 **57 61**>

Проделав эту процедуру n раз, получим отсортированный массив, затратив на это порядка O(*n* log *n*) шагов.

Если содержимое исходного массива *S* не является сортдеревом, оно может быть приведено в это состояние следующим алгоритмом. Начнём с элемента – крайнего справа узла дерева, имеющего хотя бы одного сына. Очевидно, что это элемент ⎣*n*/2⎦. Обменяем вес в этом узле с большим из весов сыновей, чтобы удовлетворить требованию к сортдереву. Выполним это действие для каждого узла, двигаясь к началу массива. Поднявшись на верхние уровни, будем наводить порядок в каждом узле рекурсивно, как и при сортировке. Обработка каждого узла тем самым потребует не более чем O(log *n*) шагов. В обработке участвует *n*/2 узлов. Следовательно, дерево сортировки может быть получено из произвольного массива за время не более чем O(*n* log *n*).

Таким образом, произвольный массив может быть отсортирован в два этапа, каждый сложностью O(*n* log *n*), т. е. всего за O(*n* log *n*) шагов. Алгоритм известен под названием сортировка деревом или пирамидальная.

На основе сортдерева реализован контейнер-адаптер «очередь с приоритетами» — *priority\_queue*<*T*>. Он использует в качестве базы *vector* и поддерживает функции-члены *push*( ) — поместить в очередь, *top*( ) — выдать наибольший и *pop*( ) — убрать наибольший. Для сравнения элементов по умолчанию используется компаратор *less*<*T*>. Если передать в шаблон компаратор *greather*<*T*>, очередь будет выдавать наименьший элемент.

## 26.2. Сортировка слиянием

Слияние – это объединение двух упорядоченных последовательностей с сохранением упорядоченности. На основе этого алгоритма можно выполнить сортировку произвольной последовательности в массиве A мощностью n. Для этого последовательность делится пополам, обе половины сортируются и затем сливаются. Сортировка для половин выполняется по той же схеме. Деление пополам продолжается до тех пор, пока не получатся последовательности размером в 1 элемент.

**Алгоритм**

void LinkSort(int i, int j) {  
 if (i < j) {  
 m = (i + j – 1) / 2;  
 LinkSort(i, m);  
LinkSort(m+1, j );  
Merge (i, m, j);   
 }  
}

Алгоритм слияния (используется буфер *B* размером *n*):

void Merge (int i, int m, int j)  
{ int i1 = i, j1 = m+1, k = 0;  
 while ( i1 < = m) && (j1 < = j))  
 B[k++] = S[i1] < S[j1]? S[i1++] : S[j1++];  
 while (i1 <= m) B[k++] = S[i1++];  
 while ( j1 <= j) B[k++] = S[j1++];  
 for (i1 = i, j1 = 0; j1 < k; ) S[i1++] = B[j1++];  
 }  
}

*Пример* 26.1. Применение сортировки слиянием.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (1) | 19 | 22 | 1,5 | 54 | 7,3 | 32 | 8 | 61 | 14 | 33 | 31 | 57 | 6 |
| (2) | 19 22 | | 1,5 54 | | 7,3 32 | | 8 61 | | 14 33 | | 31 57 | | 6 |
| (3) | 1,5 19 22 54 | | | | 7,3 8 32 61 | | | | 14 31 33 57 | | | | 6 |
| (4) | 1,5 7,3 8 19 22 32 54 61 | | | | | | | | 6 14 31 33 57 | | | | |
| (5) | 1,5 6 7,3 8 14 19 22 31 32 33 54 57 61 | | | | | | | | | | | | |

Количество уровней разбиения исходной последовательности — O(log *n*).

Сумма количеств шагов по каждому уровню — O(*n*).

Сложность: O (*n* log *n*).

Идея слияния часто используется для внешней сортировки — в случае больших данных, не помещающихся в оперативной памяти.

## 26.3. Быстрая сортировка

*Данные*: последовательность *S* = <*s*1, …, *sn*>

*Результат*: то же, последовательность упорядочена.

void Quicksort(int f, int g) {

if (g <= f) return;

(Выбрать произвольный элемент *a* из *S* );

(Разделить последовательность *S* на три части:

*S*1 = *S*[*f*…*p*] с элементами меньше *a*, *S*2 – равными *a*,

*S*3 = *S*[*q*…*g*] – больше *a* );

*Quicksort*(*f*, *p*); *Quicksort*(*q*, *g*);

}

Временная сложность: O (*n* log *n*) в среднем при условии равновероятного выбора элемента *a*. В худшем случае сложность может возрасти до O(*n*2).

Детали реализации

1. Выбор *a*. Если брать определенный элемент, например, всегда первый – получаем оценку сложности O(*n*2). Можно генерировать случайное число *i*  из интервала [1…*n*] и брать *a* = *si*. Более простой способ: взять выборку элементов из *S* и использовать медиану (можно рассмотреть, например, 1‑й, средний и последний элементы).

2. Разбиение. Удобно реализовать все этапы сразу. Так как функция вызывается рекурсивно, аргумент всегда будет частью исходной последовательности *S*[ *f* ], *S*[ *f*+1 ], … *S*[*g*]. Выбрав произвольный элемент *S*[ *i* ] = *a*, можно разбить в этом месте.

Можно использовать следующий алгоритм разбиения с помощью рабочих индексов *i*, *j*:

int i = f, j = g;  
while (i <= j) {  
while ((S[j] >= *a* ) && (j >= f) -- j;  
while ((S[i]< *a* ) && (i <= g) ++i;  
if (i < j) std::swap(S[i++], S[j--]);  
}

Замечание: (*f*, *g*) — аргументы *Quicksort*, номера элементов глобального массива *S.*

*Пример* 26.2. Проход алгоритма быстрой сортировки для *a* = 14. Выделены элементы, меняющиеся местами.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (1) | **19** | 22 | 1,5 | 54 | 7,3 | 32 | 8 | 61 | 14 | 33 | 31 | 57 | **6** |
| (2) | 6 | **22** | 1,5 | 54 | 7,3 | 32 | 8 | 61 | **14** | 33 | 31 | 57 | 19 |
| (3) | 6 | 14 | 1,5 | **54** | 7,3 | 32 | **8** | 61 | 22 | 33 | 31 | 57 | 19 |
| (4) | 6 | 14 | 1,5 | 8 | **7,3** | **32** | 54 | 61 | 22 | 33 | 31 | 57 | 19 |

## 26.4. Цифровая сортировка

Рассмотрим последовательность целых <*s*1, …, *si*, …, *sn*> , *si* ∈ [0, *m*–1]. Если значение m не слишком велико, последовательность можно упорядочить, используя следующий алгоритм:

 — организовать *m* пустых очередей – по одной для каждого из чисел 0 ÷ *m*–1 – черпаков;

 — просматривая последовательность <*s*1, …, *sn*> слева направо, поместить каждый её элемент *si*в очередь с номером, равным значению этого элемента (в очередь № *si*);

 сцепить полученные очереди в порядке возрастания номеров.

*Пример* 26.2. Карманная сортировка.

*m* = 4, *S* = <1, 2, 1, 0, 3, 2 ,1 ,3, 0, 2, 2, 3, 1, 2, 1, 0, 3, 3, 2, 1, 3, 0, 3, 1, 0 >

Просмотр последовательности и раскладывание по карманам:

0>0, 0, 0, 0, 0  
1>1, 1, 1, 1, 1, 1, 1  
2>2, 2, 2, 2, 2, 2  
3>3, 3, 3, 3, 3, 3, 3

*Результат*: 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3

Этот алгоритм известен под названием сортировка вычерпыванием (карманная). Оче­вид­но, что его временная сложность O(*n*). Она определяется пунктом 2 ал­го­рит­ма, поскольку пункты 1 и 3 выполняются за постоянное время (не за­висящее от *n*).

Алгоритм цифровой или поразрядной сортировки является естественным об­общением алгоритма сортировки вычерпыванием для кортежей.

Исходное множество представляется как декартово произведение

S = *S*1 × *S*2 × … × *Sp* и |*Si*| = *mi*.

Тогда каждый элемент сортируемой последовательности — это кортеж <*a*1,… *ai*,… , *ap*>, причём ∀*i*, *si*∈[0, *mi*–1] и (*s*1, … , *sp*) ≤ (*t*1, … , *tq*) ⇒ выполнено одно из условий:

∀*j*, *sj*  < *tj* и ∀ *i* < *j*, *si* = *ti*;

*p* ≤ *q* и *si* = *ti* , *i* ∈ [1, *p*].

Будем считать для простоты, что ∀*i*, *mi*= *m*.

Последовательность кортежей можно отсортировать за *k* проходов: сначала по *k*‑той позиции, потом по (*k* – 1)-й, и т. д. до 1‑й.

**Алгоритм**

*Данные*: последовательность *S* в виде кортежей <*ai*1, …, *aim*>

*Результат*: то же, последовательность упорядочена.

class my\_turple{ //Данные  
public:  
int pos(int p); //Извлечь позицию p  
}

int main( )  
{ queue<my\_turple> QUEUE;  
 {поместить исходную последовательность <*s*1*…s*N> в очередь *QUEUE*};  
 queue<my\_turple>Q[ m ]; //Создать черпаки  
 for (j = k – 1; k > 0; -- k) {  
 while (!QUEUE.empty( )) {  
 my\_turple A = QUEUE.front( );   
// взять кортеж *A* = < *ai*1, …, *aiN* > из очереди  
 QUEUE.pop( )   
 Q[A.pos( j )].push(A); // поместить его в черпак № (*aij*)  
 }  
 for (L = 0; L < m – 1; ++ L) // воссоздать очередь QUEUE сцеплением  
 {QUEUE ⇐ Q[L];} // черпаков в порядке возрастания их номеров  
 }  
}

Временная сложность: O ((*m*+*n*)\**k*) = O (*n*).

*Пример* 26.3. Сортировка кортежей.

Пусть *m* = 4, *k* = 3.

S = <312, 322, 032, 011, 100, 123, 302, 113, 101, 301, 210, 331, 001, 212>

Первый проход — по младшей цифре.

0>10**0**, 21**0**  
1>01**1**, 10**1**, 30**1**, 33**1**, 00**1**  
2>31**2**, 32**2**, 03**2**, 30**2**, 21**2**  
3>12**3**, 11**3**

Второй проход — по средней цифре.

0>1**0**0, 1**0**1, 3**0**1, 0**0**1, 3**0**2  
1>2**1**0, 0**1**1, 3**1**2, 2**1**2, 1**1**3  
2>3**2**2, 1**2**3  
3>3**3**1, 0**3**2

Третий проход — по старшей цифре.

0>**0**01, **0**11, **0**32  
1>**1**00, **1**01, **1**13, **1**23  
2>**2**10, **2**12  
3>**3**01, **3**02, **3**12, **3**22, **3**31

## 26.5. Cортировка цепочек

Кластеры, образующие входную последовательность, могут различаться по длине. Этот случай известен как задача сортировки цепочек. Типичный пример таких цепочек: идентификаторы в программе на каком‑нибудь алгоритмическом языке. В этом случае можно также применить рассмотренный выше алгоритм, если дополнить короткие кластеры до максимальной длины некоторым спе­ци­аль­ным значением (символом), играющим роль заведомо меньшего любых зна­че­ний, которые могут встретиться в качестве элемента кластера. В случае с иден­тификаторами для этой цели чаще всего используется символ пробела. Однако различие в длине обрабатываемых цепочек может оказаться существенным и стандартный подход приведет к большому объему лишней вычислительной работы.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* | \* | \* | \* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Рис. 26.12*. Типовая схема исходных данных для сортировки цепочек.

Потери можно исключить, если модифицировать алгоритм поразрядной сортировки следующим образом:

— сформировать из исходной последовательности вспомогательный набор *LENGTH* из *k* очередей, в каждую из которых собрать цепочки одинаковой длины (*k* ­– максимальная длина входной цепочки);

— присоединять перед каждым проходом в начало входной очереди *QUEUE* очередь *LENGTH*[ *j* ] (*j* – номер обрабатываемого элемента в кластере = длина включаемых в обработку цепочек).

Такой подход позволит отказаться как от дополнения коротких цепочек до максимальной длины, так и от пустой обработки этих дополнений.

Можно оптимизировать также процесс объединения черпаков. В процессе предварительного просмотра исходной последовательности можно сфор­ми­ро­вать *BUSY*[ *j* ] – набор множеств черпаков, которые бу­дут не пусты в *j*‑том проходе сортировки. Эта структура, например, может пред­ставлять собой матрицу размером *m*\**k* из 0 и 1: *BUSY*[ *i* ][ *j* ] = 1 означает, что значение *i* действительно встречается в позиции *j*.

**Алгоритм**

*Данные*: последовательность в виде кортежей <*ai*1, …, *aim*> разной длины.

*Результат*: то же, последовательность упорядочена.

int main( )  
{ queue <my\_turple> QUEUE, LENGTH[m];  
 int BUSY[m][k];   
 {сформировать очереди LENTH[L] и множества BUSY[L]};  
 for (j = k – 1; k > 0; -- k) {  
 {присоединить LENTH[j] к началу QUEUE};  
 queue <my\_turple> Q[m];  
 while (!QUEUE.empty( )) {  
 //< *a*i1, …, *a*ij, …, *a*iN > ⇐ QUEUE; // взять кортеж из очереди  
 my\_turple A (QUEUE.front( )); QUEUE.pop( );  
 //Q[*a*ij] ⇐ < *a*i1, …, *a*iN >; // поместить его в черпак № (*a*ij)  
 Q[ *a*ij ].push(A);  
 }  
 for (auto L : BUSY[j]) // воссоздать очередь QUEUE сцеплением  
 {QUEUE ⇐ Q[L];} // черпаков, указанных в множестве BUSY[L]  
 }  
}

*Пример* 26.4. Сортировка цепочек.

Пусть m = 4,

*S* = <3, 01, 123120, 002, 1313, 03213, 123210, 12, 33, 3210, 22131, 320123>

Шаг 1. Создаём очереди по убыванию длин цепочек.

1>3  
2>01, 12, 33  
3>002  
4>1313, 3210  
5>03213, 22131  
6>123120, 123210, 320123

Шаг 2. Просматриваем очередь 6, распределяем по 6-й позиции.

0>12312**0**, 12321**0**  
1>  
2>  
3>32012**3**

Шаг 3. Просматриваем очередь 5 и результат предыдущего шага, распределяем по 5-й позиции.

0>   
1>2213**1**, 1232**1**0  
2>1231**2**0, 3201**2**3  
3>0321**3**

Шаг 4. Очередь 4 + предыдущий результат, 4-я позиция.

0>321**0**1>123**1**20, 320**1**23, 032**1**3  
2>123**2**10  
3>131**3**, 221**3**1

Шаг 5. Очередь 3, 3-я позиция.

0>32**0**123  
1>32**1**0, 13**1**3, 22**1**31  
2>00**2**, 03**2**13  
3>12**3**120, 12**3**210

Шаг 6. Очередь 2, 2-я позиция.

0>0**0**2  
1>0**1**2>1**2**, 3**2**0123, 3**2**10**,** 2**2**131, 1**2**3120, 1**2**3210  
3>3**3**, 1**3**13, 0**3**213

Шаг 7. Очередь 1, 1-я позиция.

0>**0**02, **0**1, **0**3213  
1>**1**2, **1**23120, **1**23210, **1**313  
2>**2**2131  
3>**3**, **3**20123, **3**210**, 3**3

Результат:

*S*\*=<002, 01, 03213, 12, 123120, 123210, 1313, 22131, 3, 320123, 3210, 33>

Очевидно, что все предложенные улучшения не влияют на общую оценку временной сложности этого алгоритма, которая остаётся O (*n*).

Линейная сложность карманной, цифровой и поразрядной сортировки получается благодаря отказу от попарного сравнения элементов сортируемой последовательности. Границей временной сложности снизу в этом случае будет Ω (*n*).

*Контрольные вопросы*

1. Если элементы последовательности можно сравнивать только целиком, алгоритм сортировки какой сложности является оптимальным?
2. Какой из алгоритмов сортировки требует минимума дополнительной памяти?
3. Какой из алгоритмов сортировки лучше всего подходит для очень больших объёмов данных?
4. Как следует сортировать цепочки, если они сильно различаются по длине?
5. Сколько нужно создать черпаков для сортировки вычерпыванием следующей последовательности: <23, 34, 1, 55, 16, 10, 11, 43, 98, 35, 17, 19, 81, 22 >?
6. Какой элемент окажется в начале результата сортировки следующей последовательности цепочек десятичных цифр: <23, 34, 9, 55, 16, 1000, 11, 403, 980, 35, 107, 19, 810, 2200 >?
7. Какой элемент окажется в конце результата сортировки следующей последовательности цепочек десятичных цифр: <23, 34, 9, 55, 16, 1000, 11, 403, 980, 35, 107, 19, 810, 2200 >?

# 27. Алгоритмы на графах с нагруженными рёбрами (Кратчайшие пути и смежные вопросы)

Рассматриваются орграфы *G* = <*V*, *E*>, дугам которых приписаны веса: ∀ <*u*, *v*> ∈ *E* приписан *a*(*u*, *v*) – вес. Полагаем *a*(*u*, *v*) = ∝, если <*u*, *v*> ∉ *E*.

Если <*v*0, …, *vp*> – путь в *G*, то его длина – это сумма весов составляющих его рёбер < *vi*–1, *vi* >

.

Замечание: если все *a* = 1, имеем обычное определение длины пути как числа дуг; как и ранее, принимаем путь = 0 при *p* = 0.

Нас интересует кратчайший путь между фиксированными вершинами *s*, *t* ∈ *V*. Его длину обозначим *d*(*s*, *t*) и назовем расстоянием от *s* до t (оно может быть < 0). Если не существует пути из *s* в *t*, *d*(*s*, *t*) = ∝.

Если любой контур графа имеет неотрицательную длину, кратчайший путь не будет содержать повторов (элементарный путь). Если имеется контур отрицательной длины, обходя его достаточное количество раз, можно сделать длину пути меньшей любого наперёд заданного вещественного числа (задача не определена). Если же в этом случае говорить об определении длины эле­мен­тар­но­го пути, задача усложняется (она будет содержать в себе задачу су­ще­ство­ва­ния гамильтонова пути).

Практическая интерпретация:

— вершины – города, рёбра – дороги с длиной, указанной весом;

— то же, вес – стоимость передачи информации;

— то же, вес – вероятность безаварийной работы канала *p*(*u*, *v*). Если надёжности каналов независимы, определение самого надёжного пути сводится к задаче о кратчайшем пути с весами *a*(*u*, *v*) = log *p*(*u*, *v*);

— сетевое планирование: отыскание самого длинного пути.

Кратчайший путь можно построить, если известно расстояние между вершинами. Если нет отрицательных контуров, то ∀ *s*, *t* ∈ *V*, *s* ≠ *t* ∃ *v*∈ *V* такая, что *d*(*s*, *t*) = *d*(*s*, *v*) + *a*(*v*, *t*). Это – предпоследняя вершина на кратчайшем пути. Далее можно отыскивать *u* – предпоследнюю вершину на пути в *v*.

В итоге получаем последовательность <*s*, …, *u*, *v*, *t*> вершин на кратчайшем пути.

**Алгоритм нахождения кратчайшего пути   
по известным расстояниям**

*Данные*: *D*[*v*] – расстояния от фиксированной вершины *v* до всех остальных, *v* ∈*V*;

*t* – фиксированная вершина;

*A*[*u*][*v*] – вещественная матрица весов рёбер, *u*, *v* ∈*V*.

*Результат*: последовательность вершин на кратчайшем пути в стеке *STACK*.

main ( )  
{ stack <int> STACK;  
 STACK.push(t); v = t;  
 while (v != s) {  
u = (вершина, для которой abs(D[v] – (D[u] + A[u][v]) ) < ϵ; (\*)  
 STACK.push(u);  
 v = u;  
 }  
}

Поскольку веса и расстояние — вещественные числа, сравнение их на равенство не допускается. Нужно проверять то, что модуль разности меньше заданного ϵ > 0, определяющего точность измерений.

*Пример* 27.1. Нахождение кратчайшего пути по матрице весов и вектору расстояний.

|  |  |
| --- | --- |
|  |  |
| *D* = [0, 1, 4, 3, –1] |
| *s* = 1, *t* = 5;  кратчайший путь: <1, 2, 3, 5> |

*Рис. 27.1.*

Для орграфа <*V*, *E*>, |*V*| = *n*, |*E*| = *m*, если в (\*) происходит перебор всех вершин (по матрице *A*[*u*][*v*]), сложность O(*n*2). Если используются списки предшествования *PREV*[*v*], сложность O(*m*).

Примечание. Списки (множества) предшествования *PREV*[*v*] — это альтернатива спискам смежности *LIST*[*v*], которую можно использовать только для орграфов. Множества *LIST*[*v*] — это вершины — концы дуг, выходящих из *v*, а *PREV*[*v*] — множества вершин, из которых имеются рёбра с концами в данной вершине. Списки предшествования обеспечивают проверку не всех (*n*–1) дуг, могущих входить в вершину, а только фактически имеющихся в графе. Сумма количеств таких дуг (полустепеней захода) по всем обрабатываемым вершинам будет O(*m*).

Для неорграфа в случае положительных весов задача сводится к задаче для орграфа заменой каждой дуги {*u*, *v*} парой дуг <*u*, *v*> и <*v*, *u*> с одним и тем же весом. При наличии отрицательного веса такая замена даёт контур с от­ри­ца­тель­ной длиной. Поэтому для в задачах на кратчайшие пути в неорграфе допускается использовать только неотрицательные веса рёбер.

Далее:

— рассматриваем только орграфы;

— предполагаем, что *m* = Ω(*n*) (*n* = O(*m*)), то есть исключается случай «большинство вершин изолировано»;

— веса дуг хранятся в массиве *A*[*u*][*v*] — матрице весов *A*[*u*, *v*].

*Вывод*. Для построения кратчайшего пути между произвольной парой вершин *s*, *t* в ориентированном нагруженном графе *G* = <*V*, *E*>, заданном матрицей весов рёбер *A*[*u*, *v*], достаточно вычислить вектор расстояний *D*[*v*] от стартовой вершины *s* до всех остальных вершин. Поэтому все рассматриваемые далее алгоритмы отыскания кратчайшего пути на самом деле вычисляют только вектор расстояний.

# 27.1. Кратчайший путь от фиксированной вершины. Общий случай

Идея большинства алгоритмов вычисления расстояния d(s, t):

— для заданной матрицы весов A[u, v] вычисляем некоторые верхние ограничения D[v] для расстояний от s до v∈ V;

— каждый раз при обнаружении вершины u∈ V, для которой

D[u] + A[u, v] < D[v], (\*\*)

производим улучшение оценки

D[v] = D[u] + A[u, v,];

— если дальнейшее улучшение оценок невозможно, то это означает, что вектор *D*[*v*] содержит значения расстояний *d*(*s*, *v*).

Таким образом, для определения расстояния *d*(*s*, *t*) нужно получить вектор расстояний *d*(*s*, *v*) для всех *v*∈ *V*. Неизвестен более простой способ.

Эффективность алгоритма зависит от очерёдности выбора *u* и *v* при про­вер­ке условия (\*\*).

В общем случае нужно пытаться улучшить оценку расстояния, проверяя все комбинации *u* и *v*. Такой подход известен как

**Алгоритм Форда-Бэллмана**

*Данные*: орграф *G* = <*V*, *E*>, заданный в форме вещественной весовой матрицы *A*[*u*][*v*], не содержащий циклов с отрицательной длиной, и вершина-источник *s*.

*Результат*: *D*[*v*] – вектор расстояний *d*(*s*, *v*).

int main ( )  
{ for (v : V) D[v] = A [s][v];  
 // D[s] = 0; – если нужно!  
 for ( int k = 1; k < n – 1; ++k ) //Цикл итераций  
 // (кратчайший путь содержит не более (*n*–1) рёбер)   
 for (v : (V \ { s })) //Пересчёт оценок расстояний  
 for (u : V) D[v] = min ( D[v], D[u] + A[u][v] );  
}

Алгоритм в качестве начального значения для вектора расстояний использует строку матрицы весов. Такая оценка является не чем иным как оценкой расстояния на пути из одного ребра. Это может быть вес ребра (если оно есть) или ∞ (ребра между вершинами нет). Улучшение оценки представляет собой замену пути из *m* рёбер путём из (*m* +1) ребра (*m* = 1, 2, …). Очевидно, что путь в графе, не являющийся циклом, может состоять не более чем из (*n* – 1) рёбер. Следовательно, для получения вектора расстояний может потребоваться не более (*n* – 2) итераций. Фактически итерации можно прекратить, как только вектор *D* перестанет изменяться.

*Задание*: предложить соответствующую модификацию алгоритма.

Очевидно, что временная сложность алгоритма Форда-Бэллмана – O(*n*3). Возможная модификация не меняет этой оценки.

Если рёбер в графе относительно немного (*m* << *n*2), алгоритм можно модифицировать заменой последней строки на строку

for (u : PREV[v]) D[v] = min ( D[v], D[u] + A[u][v] ),

т. е. использовать дополнительную структуру данных – списки пред­ше­ство­вания. Временная сложность при этом будет O(*n*\**m*).

*Пример* 27.2. Тест алгоритма Форда-Беллмана

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Рис. 27.2.* | ∝ | 1 | ∝ | ∝ | 3 |
| ∝ | ∝ | 3 | 3 | 8 |
| ∝ | ∝ | ∝ | 2 | -5 |
| ∝ | ∝ | ∝ | ∝ | 1,5 |
| ∝ | ∝ | ∝ | 4 | ∝ |
| Матрица весов | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *к* | *D*[0] | *D*[1] | *D*[2] | *D*[3] | *D*[4] |
| — | 0 | 1 | ∝ | ∝ | 3 |
| 1 | 0 | 1 | 4 | 4 | –1 |
| 2 | 0 | 1 | 4 | 3 | –1 |
| 3 | 0 | 1 | 4 | 3 | –1 |

### 27.2. Алгоритм Дейкстры

*Данные*: орграф *G* = <*V*, *E*>, заданный в форме вещественной весовой матрицы *A*[*u*, *v*], не содержащий циклов с отрицательной длиной, и вершина-источник *s*.

Веса не отрицательны.

*Результат*: vector <double> D{n} – вектор расстояний *d*(*s*, *v*).

Если *D*[*p*] = min (*D*[*v*]: *v* ∈ *V* \ {*s*}), то *D*[*p*] — расстояние. Вершину *p* можно исключить и все остальные оценки пересчитать через *A*[*p*][*v*].

int main ( )  
{ for (auto v : V) D[v] = A [s][v];  
 // *D*[*s*] = 0; – если нужно!  
 T = V\{s};  
 while (T != ∅) {  
 u = {p: D[p] = min{D[q]), q ∈ T};   
 T = (T\{u});  
 for (auto u : T) D[v] = min ( D[v], D[u] + A[u][v] );  
}

Для выполнения алгоритма нужно O(*n*) раз просмотреть массив *D* из *n* элементов, т. е. алгоритм Дейкстры имеет квадратичную сложность: O(*n*2).

Множество *T* можно представить как массив из номеров вершин от 0 до (*n*–1) включительно. После формирования *D*[*v*] обнуляем *D*[*s*] и меняем в векторе *Т* местами *T*[0] и *T*[*s*], а в векторе *D* — соответственно *D*[0] и *D*[*s*].

Далее в цикле просмотр вектора *D* начинаем с *D*[*u*], *u* = 1, 2,… (*n*–1); на каждом шаге цикла находим *p*, меняем местами *T*[*u*] и *T*[*p*], *D*[*u*] и *D*[*p*]. Пересчёт ведём, используя вектор *T* для перенумерации: *T*[*v*] вместо *v*.

for (int v = 0; v < n; ++v) { D[v] = A[s][v]; T[v] = v; }  
D[s] = 0; if(s) { swap(D[0], D[s]); swap(T[0], T[s]); }  
int p, q, r = 1;  
while(r < n) {  
 double d = ∝;  
 for(p = r; p < n; ++p)  
 if(d < D[p]) { d = D[p]; q = p; }  
 if(q != r) { swap(T[q], T[r]); swap(D[q], D[r]); }  
 for(u = r+1; u < n; ++u) D[T[u]] = min(D[T[u]], D[T[r]]+A[T[r]][T[u]]);  
 ++r;  
}

*Пример* 27.3. Работа алгоритма Дейкстры

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Рис. 27.3.* | Матрица весов | | | | | |
| 0 | 1 | ∝ | ∝ | ∝ | ∝ |
| ∝ | 0 | 5 | 2 | ∝ | 7 |
| ∝ | ∝ | 0 | 1 | ∝ | 1 |
| ∝ | 2 | ∝ | 0 | 4 | ∝ |
| ∝ | ∝ | ∝ | 3 | 0 | ∝ |
| ∝ | ∝ | ∝ | ∝ | 1 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| шаг | *D*[1] | *D*[2] | *D*[3] | *D*[4] | *D*[5] | *D*[6] |
| 0 | 0 | **1** | ∝ | ∝ | ∝ | ∝ |
| 1 | 0 | 1 | 6 | **3** | ∝ | 8 |
| 2 | 0 | 1 | **4** | 3 | 7 | 8 |
| 3 | 0 | 1 | 4 | 3 | 7 | **5** |
| 4 | 0 | 1 | 4 | 3 | **6** | 5 |

### 27.3. Кратчайший путь в бесконтурном графе

В бесконтурном графе можно перенумеровать вершины так, что

∀ <*u*, *v*>∈ *T* => *Num*[*u*] < *Num*[*v*]

Исходные данные: орграф в виде списков смежности *LIST* и списков предшествования *PREV*.

vector<list<double≫LIST {n}, PREV {n};

Шаг 1. Перенумерация вершин, чтобы каждое ребро вело от меньшего номера к большему (топологическая сортировка).

stack<int>STACK;  
vector <int> In {n, 0};  
for(int v = 0; v < n; ++v)  
 for (auto u : NEXT[ v ]) ++In[ u ]; //счёт степеней захода  
for(int v = 0; v < n; ++v) if(!In[ v ]) STACK.push(v);   
int num = 0;  
while (!STACK.empty( )) {  
 int u = STACK.top(); STACK.pop();  
 Num[u] = ++num;  
 for (auto u : NEXT[v]) {  
 – –In[ u ];  
 if( !In[ u ] ) STACK.push( u );  
 }  
}

Шаг 2. Отыскание вектора расстояний *D*.

vector <double> D {n, 0};  
// D[Num[0]] = 0;  
for(int v = 1; v < n; ++v) {  
 D[ Num[ v ] ] = ∝;  
 for (auto u : PREV[ Num[ v ] ])  
 D[ u ] = min(D[ u ], D[ Num[ v ] ] + A[ u ][ Num[ v ] ]);  
Оба шага алгоритма имеют сложность O(*m*).

*Пример* 27.4. Перенумерация и кратчайший путь в бесконтурном графе



*Рис. 27.4.* Кратчайший путь в бесконтурном графе

### 27.4. Алгоритм Флойда

*Дано*: непустой взвешенный гpаф *G* = <*V*, *E*> с произвольными весами рёбер (дуг). Требуется найти длины кратчайших путей (расстояния) между всеми парами вершин графа, если в графе нет циклов (контуров) отрицательной суммарной длины, либо обнаружить наличие таких контуров.

*Инициализация*:

1. Построим матрицу *D*0 размерности |*V*| × |*V*|, элементы которой определяются по правилу (инициализация матрицей весов):

*dii*0 = 0;

*dij*0= Вес (*vi*, *vj*), *i* ≠ *j*, если в графе существует ребро (дуга) (*vi*, *vj*);

*dij*0= ∞, *i* ≠ *j*, если нет ребра (дуги) (*vi*, *vj*).

2. *m* = 0.

3. Построим матрицу *Dm*+1 по *Dm*, вычисляя её элементы следующим образом:

*dijm*+1 = min{*dijm*, *di*(*m*+1)*m* + *d*(*m*+1)*jm*}, *i* ≠ *j*; *diim*+1=0 . (\*)

Если *dimm* + *dmim* < 0 для какого-то *i*, то в графе существует цикл (контур) отрицательной длины, проходящий через вершину *vi*.

4. *m* = *m* + 1; пока *m* < |*V*|, повторяем шаг (3).

*Результат*: элементы последней построенной матрицы *D*|*V*| равны расстояниям между соответствующими вершинами.

Если требуется найти сами пути, то перед началом работы алгоритма построим матрицу *P* с начальными значениями элементов *pij* = *i*. Каждый раз, когда на шаге (1) значение *dijm*+1 будет уменьшаться в соответствии с (\*) (т. е. когда *di*(*m*+1)*m* + *d*(*m*+1)*jm* < *dijm* ), выполним присваивание *pij* = *p*(*m*+1)*j*. В конце работы алгоритма матрица *P* будет содержать данные для определения кратчайшего пути между любой парой вершин: значение *pij* будет равно номеру предпоследней вершины на пути между *i* и *j* (если *pij* = *i*, путь не существует).

*Примечаниe*: если граф – неориентированный, то все матрицы *Dm* являются симметричными, поэтому достаточно вычислять элементы, находящиеся только выше (либо только ниже) главной диагонали.

for(int i = 0; i < n; ++i) {  
 for(int j = 0; j < n; ++j) { D[ i ][ j ] = A[ i ][ j ]; P[ i ][ j ] = 0; }  
 D[ i ][ i ] = 0;  
}  
for(int m = 0; m < n; ++m)  
 for(int i = 0; i < n; ++i) {  
 for(int j = 0; j < n; ++j)   
 if(D[ i ][ j ] > D[ i ][ m ] + D[ m ][ j ]) {  
 D[ i ][ j ] = D[ i ][ m ] + D[ m ][ j ]); P[ i ][ j ] = m;

**Выдача пути по матрице промежуточных вершин *P*[ *n* ][ *n* ]:**

void OutTrek(int p, int q) {  
 if(P[ p ][ q ]) { OutTrek(p, P[ p ][ q ]); OutTrek(P[ p ][ q ], q); }   
 else { cout << " " << p; if(p != q) cout << q; } }

*Пример* 27.5. Испытание алгоритма Флойда

Матрица весов

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **1** | 0 | — | 4 | — | — | 17 | — | 39 |
| **2** | — | 0 | 6 | — | 31 | — | 5 | — |
| **3** | 20 | 17 | 0 | 2 | 20 | 12 | 6 | — |
| **4** | — | — | — | 0 | — | 3 | — | — |
| **5** | — | 16 | — | — | 0 | — | — | — |
| **6** | 27 | 34 | 9 | 2 | 35 | 0 | — | 9 |
| **7** | 25 | — | — | — | — | — | 0 | — |
| **8** | 33 | 19 | — | — | — | 25 | 13 | 0 |

Матрица расстояний Матрица промежуточных вершин

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **1** |  | 31 | 4 | 6 | 24 | 9 | 10 | 18 |  | 0 | 3 | 0 | 3 | 3 | 4 | 3 | 6 |
| **2** | 26 |  | 6 | 8 | 26 | 11 | 5 | 20 |  | 3 | 0 | 0 | 3 | 3 | 4 | 0 | 6 |
| **3** | 20 | 17 |  | 2 | 20 | 5 | 6 | 14 |  | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 6 |
| **4** | 30 | 29 | 12 |  | 32 | 3 | 18 | 12 |  | 6 | 6 | 6 | 0 | 6 | 0 | 6 | 6 |
| **5** | 42 | 16 | 22 | 24 |  | 27 | 21 | 36 |  | 3 | 0 | 2 | 3 | 0 | 4 | 2 | 6 |
| **6** | 27 | 26 | 9 | 2 | 29 |  | 15 | 9 |  | 0 | 3 | 0 | 0 | 3 | 0 | 3 | 0 |
| **7** | 25 | 46 | 29 | 31 | 49 | 34 |  | 43 |  | 0 | 3 | 1 | 3 | 3 | 4 | 0 | 6 |
| **8** | 33 | 19 | 25 | 27 | 45 | 25 | 13 |  |  | 0 | 0 | 2 | 3 | 3 | 0 | 0 | 0 |



*Рис. 27.5*. Результат построения кратчайших путей по матрице расстояний

### 27.5. Транзитивное замыкание отношения (алгоритм Уоршалла)

Матрица смежности ориентированного графа задаёт бинарное отношение на множестве вершин. Это отношение можно дополнить, добавив рёбра между каждой парой вершин, связанных путём. В результате такой операции бинарное отношение получает свойство транзитивности.

Для выполнения транзитивного замыкания бинарного отношения нужно выполнить следующие действия.

1. Инициализировать *D* матрицей смежности.

2. В тройном цикле по *n*:

for(int m = 0; m < n; ++m)  
 for(int i = 0; i < n; ++i)  
 for(int j = 0; j < n; ++j) D[ i ][ j ] = D[ i ][ j ] || D[ i ][ m ] && D[ m ][ j ];

Результат: матрица достижимости за время O(*n*3).

Вариант — чистка цикла:

if (D[ i ][ m ]) for(int j = 0; j < n; ++j) D[ i ][ j ] = D[ i ][ j ] || D[ m ][ j ];

Внутренний цикл можно заменить манипуляцией с машинными словами, получив сложность O(*n*2).

Для неорграфа получается набор полных подграфов — компонент связности, который можно получить за время O(*n*+*m*) обходом в глубину или в ширину.

*Контрольные вопросы*

1. В неорграфе с нагруженными рёбрами нет циклов отрицательной стоимости. Какой алгоритм отыскания расстояний от заданной вершины до всех остальных целесообразно применить?
2. Задан граф из 100 вершин и 1000 рёбер. Рёбра нагружены весом, не превышающим 10. Какое значение «бесконечности» достаточно использовать в качестве веса несуществующих рёбер в матрице весов?

# 28. Алгоритмы на графах с нагруженными рёбрами и вершинами. Потоки в сетях

#### 28.1. Максимальный поток в сети. Основные понятия

Сеть – это пара *S* = <*G*, *c*>, где *G* = <*V*, *E*> – произвольный ориентированный граф, а *c*: *E* → *R*+ – функция, которая ставит в соответствие каждой дуге <*u*,*v*> неотрицательное вещественное число *c*(*u*,*v*), называемое пропускной способностью этой дуги. Множества *V* и *E* будем называть множеством вершин и множеством дуг сети *S*.



*Рис. 28.1*. Пример сети. Орграф с неотрицательными весами рёбер.

Для произвольной функции

*f* : *E* → *R* + (1)

и произвольной вершины *v* сети *S* рассмотрим величину

*Divf* (*v*) = .

Если интерпретировать *f* (*u*,*v*) как поток из *u* в *v*, то величина *Divf* (*v*) определяет «количество потока», вытекающего из вершины *v*. Эта величина может быть положительной (поток «проистекает» из вершины), отрицательной (поток «на­кап­ли­ва­ется» в вершине) или нулевой.



*Рис. 28.2*. Поток в вершине 6: *Divf* (*v*6) = (2+1+2) – (5+0+0).

Выделим в сети S две вершины – источник s и сток t (s ≠ t). Потоком из s в t в сети S будем называть произвольную функцию вида (1), для которой выполняются условия

∀ <*u,v*>∈*E* ⇒ 0 ≤ *f* (*u,v*) ≤ *c*(*u,v*) (2)

и

∀ *u* ∈ *V* \ {*s,t*} ⇒ *Div f* (*v*)=0 (3)

Величину

*W* (*f*) = *Div f* (*s*)

будем называть величиной потока *f*. Таким образом, поток не возникает и не накапливается ни в какой вершине, отличной от *s*, *t*, и через любую дугу <*u*, *v*> можно пропустить не более чем *c*(*u*, *v*) единиц потока.

Рассматривается задача нахождения максимального потока (т. е. потока с максимальной величиной) в заданной сети.

Зафиксируем несколько предварительных соображений.

Разрез *P*(*A*) сети *S*, соответствующий подмножеству *A* ⊂ *V* (*A* ≠ ∅, *A* ≠ *V*) – это множество дуг  
 <*u*, *v*> ∈ *E* таких, что ­­*u* ∈ *A* и *v* ∈ *V* \ *A*, т. е. *P* (*A*) = *E* ∩ (*A*× (*V* \ *A*) .

Для произвольного потока *f* в сети *S* величина потока через разрез *P*(*A*) определяется естественным образом:



*Лемма* 1. Если­­ *s* ∈ *A* и *t* ∈ *V* \ *A*, то для произвольного потока *f* из *s* в *t*

*W*(*f*) = *f*(*A*, *V* \ *A*) ­– *f*(*V* \ *A*, *A*) (4)

Принимая *A* = *V* \ { *t* }, получаем в качестве частного случая

*Divf*(*s*) = *W*(*f*) = *f*(*V* \ {*t*}, {*t*}) – *f*({*t*}, *V* \ {*t*}) = – (*f*({*t*},*V* \ {*t*}) – *f*(*V* \ {*t*}, {*t*}) = – *Divf*(*t*),

что означает очевидный факт: в сток входит точно такое же количество потока, какое выходит из источника. Общее количество потока, таким образом, можно измерить в произвольном разрезе, отделяющем *s* от *t*.

*Пропускная способность* разреза  *P*(*A*) определяется как



Под *минимальным разрезом,* разделяющим *s* и *t*, будем понимать произвольный разрез *P* (*A*),  
*s* ∈ *A*, *t* ∈ *V* \ *A*, с минимальной пропускной способностью.

*Теорема* 2 (Форд и Фалкерсон). Величина каждого потока из *s* в *t* не превосходит пропускной способности минимального разреза, разделяющего *s* и *t*, и существует поток, достигающий этого значения.

Пусть *P*(*A*) – минимальный разрез.Из леммы 1 для произвольного потока *f* имеем

 (5)

Для максимального потока неравенство в формуле (5) переходит в равен­ство.

#### 28.2. Алгоритм построения максимального потока

Все известные алгоритмы построения максимального потока основываются на последовательном увеличении потока и опираются чаще всего на ***метод увеличивающих цепей***.

***Увеличивающей цепью***(***длины*** *l*) для данного потока *f* из  *s* в *t* называется произвольная последовательность попарно различных вершин и дуг

*v*0*, e*1, *v*1, *e*2, *v*2, …, *vl*-1, *el*, *vl*(6)

такая, что *v*0 = *s*, *vl* = *t*, и для каждого *i* ≤ *l* дуга *ei* *допустима* из *vi*-1 в *vi* относительно потока *f.*

Дуга  *e*  сети  *S*  называется ***допустимой дугой***из  *u* в *v*  относительно потока  *f*, если

*e* = <*u, v*> и *f* (e) *< c* (*e*) (7)

или

*e* = <*v, u*> и *f* (*e*) > 0 (8)

Допустимая дуга из *u* в *v* может быть ***согласованной***(7) или ***несогласованной*** (8).

Знание увеличивающей цепи вида (6) позволяет увеличить величину потока *f* на

*δ =* min ( Δ(*e*i): 1 ≤ *i* ≤ *l* ),

где Δ(*e*i) = *c* (*e*i) –  *f* (ei), если дуга ei согласована, и *f* (ei), если не согласована. Для этого достаточно увеличить поток на каждой согласованной дуге

*f’*(*e*i) *= f* (*e*i) + *δ*

и уменьшить на несогласованной

*f’*(*e*i) *= f* (*e*i) – *δ.*

Для всех прочих дуг *f’*(*e*i) *= f* (*e*i). Полученная функция  *f’* будет *потоком*, так как ∀ *i*, *Div f’* (*vi*) останется = 0, изменения на дугах в увеличивающей цепи взаимно компенсируются и 0 ≤ *f’*(*e*i) ≤  *с* (*e*i) для всех дуг. Поток увеличивается на *δ* :

*W* (*f’*) = *Div f’* (*s*) = *Div f* (*s*) + *δ* = *W* (*f*) + *δ.*



*Рис*. *28.3*. Пример увеличивающей цепи.



*Рис*. *28.4*. Поток увеличен, и цепь разорвалась (в двух местах)

*Пример*. В следующей сети, где на каждой дуге показана величина потока и рядом (в скобках) пропускная способность дуги.



*Рис*. *28.5*. Увеличивающая цепь в сети

Увеличивающей будет цепь 1, <1, 3>, 3, <2, 3>, 2, <5, 2>, 5, <5, 6>, 6, <7, 6>, 7, <7, 8>, 8. В этой цепи дуги <1, 3>, <5, 6> и <7, 8> – согласованные (Δ = 3, 1 и 2), а дуги <2, 3>, <5, 2> и <7, 6> – несогласованные (Δ = 4, 6 и 4 соответственно). *δ =* min ( Δ(*e*i) ) = 1. Поэтому для увеличения потока можно добавить 1 к потоку на согласованных дугах и вычесть 1 на несогласованных.



*Рис*. *28.6*. Результат увеличения — максимальный поток.

Дуга <5, 7> стала насыщенной, т. е. перестала быть допустимой, а рас­смот­рен­ная цепь – увеличивающей. Показанный на рисунке минимальный разрез стал насыщенным, а, следовательно, поток в сети от *s* к *t* – максимальным. Отметим, что в рассмотренную выше увеличивающую цепь после вершины 6 можно было бы включить ещё три согласованные дуги: <6, 4>, 4, <4, 5>, 5, <5, 6>. Однако очевидно, что допущенное при этом нарушение единственности вершин и дуг в увеличивающей цепи приводит к тому, что при добавлении потока на дугах в дублирующихся вершинах нарушается баланс, и по­лу­чен­ная в результате новая функция *f’* не будет потоком.

Теорема 3. Три условия эквивалентны:

(а) поток из *s* и *t* максимальный;

(б) не существует увеличивающей цепи для  *f*.

(в)  *W* ( *f* ) = *c* ( *A*, *V* \ *A* ) для некоторого *A* ⊆ *V* , *s* ∈ *A* , *t* ∉ *A*.

Максимальный поток в сети можно найти, если, начав с произвольного потока (например, с нулевого: *f* (*e*) = 0, ∀ *e* ∈ *E*), искать увеличивающие цепи и увеличивать вдоль них фактический поток. Отсутствие в сети увеличивающих цепей будет означать, что максимальный поток найден. Однако доказано, что существуют сети, в которых можно так подбирать увеличивающие цепи, что процесс никогда не закончится.

*Пример*. Злокозненный подбор увеличивающих цепей.



*Рис*. *28.7.* Исходная сеть (поток = 0) и увеличивающая цепь.



*Рис*. *28.8*. Первая итерация.



*Рис*. *28.9.* Вторая итерация.



*Рис. 28.10*. Третья итерация.

**28.3. Метод Диница**

Идея Диница заключается в отыскании кратчайших увеличивающих цепей и построении из них вспомогательной бесконтурной сети. В этой сети производится увеличение потока (отыскивается некоторый поток, не обязательно максимальный). Далее найденный поток добавляется к потоку в исходной сети, и процесс повторяется с начала до тех пор, пока будет возможно отыскивать новые увеличивающие цепи.

#### Построение вспомогательной бесконтурной сети

Вспомогательная бесконтурная сеть *S f* имеет структуру, точно отображающую все кратчайшие увеличивающие цепи из *s* в *t* относительно фактического потока *f.*  Сеть строится в графе *G f* с дугами, определяемыми допустимыми относительно фактического потока *f*  дугами исходной сети. Сеть *S f*  содержит источник *s*, сток *t* и дуги графа вида *<u, v>,* где *u* находится на расстоянии *d*, а *v* – на расстоянии *d* + 1 от *s*, 0 *d*  *l –* 1, где *l* есть *длина* сети *S f*, т. е. расстояние от *s* до *t* в графе *G f*. Пропускную способность *c f(u, v)* мы определяем как *c (u, v) – f (u, v)* или как *f (v, u)* в зависимости от того, является ли дуга *<u, v>* согласованной или нет (или как сумму этих значений, если одновременно существуют согласованная и несогласованная допустимые дуги от *u* к *v* ).

**Функция *PSA*: построение вспомогательной бесконтурной сети *S f***

*Данные*:

— *s*, *t* – источник и сток;

— *LIST*[*v*], *PREV*[*v*] – исходная сеть *S* (списки смежности и предшествования);

— *C*[*u*][*v*] – массив пропускных способностей дуг, (*u*, *v* ∈ *V*);

— *F*[*u*][*v*] – фактический поток;

— *XV*, *XLIST*, *XPREV*, *XC*, *XF* – множество вершин и другие данные для вспомогательной бесконтурной сети;

— *QUEUE* – очередь вершин для обхода в ширину;

— *D*[*v*] – массив расстояний от источника *s*;

PSA ( ) {  
 for (auto u : V) { // Инициализация  
 D[u] = ∞;  
 XPREV[u] = ∅;  
 XLIST[u] = ∅;  
 for (auto v : V) { XC[u][v] = 0; XF[u][v] = 0; }  
 }  
 queue <int> QUEUE; XV = ∅; D[ s ] = 0;  
 // Поиск в ширину от источника *s*  
 QUEUE.push( s );  
 while (!QUEUE.empty( )) {  
 u = QUEUE.front( ); QUEUE.pop( ); XV = (XV∪{u});  
 for (auto v : LIST[ u ]) // поиск согласованных дуг  
 if ((D[ u ] < D[ v ]) && (D[ v ] <= D[ t ]) && (F[ u ][ v ] < C[ u ][ v ])) {  
 if (D[ v ] = = ∞) QUEUE.push( v ); // *v* – новая  
 D[ v ] = D[ u ] + 1;  
 XLIST[ u ] = (XLIST[ u ] ∪ {v}); // добавление <*u*, *v*> к сети *S f*  
 XPREV[ v ] = (XPREV[ v ] ∪ {u});  
 XC[ u ][ v ] = C[ u ][ v ] – F[ u ][ v ];  
 }  
 for (auto v : PREV[ u ]) // поиск несогласованных дуг  
 if ((D[ u ] < D[ v ]) && (D[ v ] <= D[ t ]) && (F[ v ][ u ] > 0)) {  
 if (D[ v ] = = ∞) QUEUE.push( v ); // *v* – новая  
 D[ v ] = D[ u ] + 1;  
 if (XC[ u ][ v ] = = 0) { // добавление <*u*, *v*> к сети *S f*  
 XLIST[ u ] = (XLIST[ u ] ∪{v});  
 XPREV[ v ] = (XPREV[ v ] ∪ {u});  
 }  
 XC[ u ][ v ] += F[ v ][ u ];   
 }  
 }  
}

*Сложность*: каждая вершина v помещается в очередь и удаляется из нее один раз; каждая дуга, инцидентная v, анализируется один раз (в одном из циклов) за константное время – итого получается *O* (n + m). Однако ини­ци­а­ли­за­ция массивов XC и XF требует *O*(n2), что дает оценку *O* (n2) для всего алгоритма.

#### Построение псевдомаксимального потока во вспомогательной бесконтурной сети

Используется понятие *потенциала* вершины сети, т. е. максимального количества потока, который можно пропустить через эту вершину. Он определяется как меньшая из двух величин: максимального возможного потока в вершину (*входной потенциал*) и из нее (*выходной потенциал*).

**Функция MAXPSA: построение псевдомаксимального потока**

*Данные*:

*s*, *t* – источник и сток;

*XV*, *XC*, *XF*, *XPREV*, *XLIST* – данные для вспомогательной бесконтурной сети;

*Pin*[ *v* ], *Pout*[ *v* ], *P*[ *v* ] – входной, выходной и полный потенциалы для *v*;

*STACK* – стек для вершин с нулевым потенциалом;

*Q*[*v*] – «грузы» в вершинах.

MAXPSA( ) {  
 stack <int> STACK; // Стек для вершин с нулевым потенциалом  
 for (auto v : XV) { // Определение потенциала вершины *v* Pin[v] = Pout[v] = 0; //Входной  
 if (v = = s) Pin [v] = ∞;  
 else for (auto u : XPREV[v]) Pin[v] += XC[u][v];  
 if (v = = t) Pout[v] = ∞; //Выходной  
 else ∀ (u ∈ XLIST[v]) Pout[v] += XC[v][u];  
 P[v] = *min* (Pin[v], Pout[v]); //Полный потенциал  
 if (P[v] = = 0) STACK.push( v );  
 Q[v] = 0; // инициализация «груза» в вершине  
 }  
 XN = XV; // в *XN* отбираются вершины с ненулевым потенциалом  
 while (XN != ∅) {   
 while (!STACK.empty( )) { // удаление вершин с нулевым потенциалом  
 v = STACK.top( ); STACK.pop( ); XN = (XN \ {v});  
 for (auto u : XPREV[v]) { // удаление дуг, входящих в *v* Pout[u] – = (XC[u][v] – XF[u][v]);  
 XLIST[u] = (XLIST[u] \ {v});  
 if (P[ u ] != 0) {  
 P[u] = *min* (Pin[u], Pout[u]);  
 if (P[u] = = 0) STACK.push( u );  
 }  
 }  
 XPREV[v] = ∅;  
 for (auto u : XLIST[ v ]) { // удаление дуг, исходящих из *v*  
 Pin[ u ] – = (XC[ v ][ u ] – XF[ v ][ u ]);  
 XLIST[ v ] = (XLIST[ v ] \ {u});  
 XPREV[ u ] = (XPREV[ u ] \ {v});  
 if (P[u] != 0) {  
 P[u] = *min* (Pin[u], Pout[u]);  
 if (P[u] = = 0) STACK.push( u );  
 }  
 }  
 XLIST[v] = ∅;  
 } // while ( STACK != ∅)  
 // *XN* сформировано  
 if (XN != ∅) { // поток ещё не максимальный  
 p = ∞; // поиск вершины *r* с потенциалом *p* = min(*P*[*v*])  
 for (auto v : XN)   
 if ( P[v] < p ) { r = v; p = P[r]; }  
 // построение потока величины *p* из *r* в *t*  
 queue <int> QUEUE; QUEUE.push( r ); Q[ r ] = p;  
 do {  
 v = QUEUE.front( ); QUEUE.pop( );  
 Pin[ v ] – = Q[ v ]; Pout[ v ] – = Q[ v ]; P[ v ] – = Q[ v ];  
 if ( P[ v ] = = 0 ) STACK/push( v );  
 if (v = = t) Q[ v ] = 0; // *t* – всегда «новая»  
 else { // разгрузка вершины  
 while (Q[v] > 0) { // пока *v* не доразгружена  
 u = (первая в списке XLIST[v]);  
 if ( Q[u] = = 0) QUEUE.push( u );  
 delta = *min* (Q[ v ], XC[ u ][ v ] – XF [ u ][ v ]);  
 XF[ u ][ v ] += delta;  
 Q[ v ] – = delta; Q[ u ] += delta;  
 if (XF[ u ][ v ] = = XC[ u ][ v ]) {// удаление <*u*,*v*>  
 XLIST[ v ] = (XLIST[ v ] \ {u});  
 XPREV[ u ] = (XPREV[ u ] \ {v});   
}  
 } // while  
 } // else  
 } while ( v != t ); // поток из *r* в *t* найден

// Построение потока величины *p* из *s* в *r*.  
 QUEUE.clear( ); QUEUE.push( r ); Q[ r ] = p;  
 do {  
 v = QUEUE.front( ); QUEUE.pop( );  
 if ( v != r ) { // *P*[*v*] ещё не уменьшен  
 Pin[ v ] – = Q[ v ]; Pout[ v ] – = Q[ v ]; P[ v ] – = Q[ v ];  
 if ( P[ v ] = = 0) STACK.push( v );  
 }  
 if ( v = = s ) Q[v] = 0; // *s* – всегда «новая»  
 else { // разгрузка вершины *v*  
 while (Q[ v ] > 0) {  
 u = (первая в списке XPREV[ v ]);  
 if ( Q[ u ] = = 0) QUEUE/push( u );  
 delta = *min* (Q[ v ], XC[ u ][ v ] – XF [ u ][ v ]);  
 XF[ u ][ v ] += delta;  
 Q[ v ] – = delta; Q[ u ] += delta;  
 if (XF[ u ][ v ] = = XC[ u ][ v ]) { //удаление <*u*,*v*>  
 XPREV[ v ] = (XPREV[ v ] \ {u});  
 XLIST[ u ] = (XLIST[ u ] \ {v}); }  
 } // while  
 } // else  
 } while ( v != s ); // найден поток из *s* в *r*  
 } // while ( XN != ∅) (главный цикл)  
}

*Оценка сложности*:

Определение потенциалов требует *O*(*n*+*m*) шагов, т. к. в цикле по вершинам каждая дуга анализируется не более двух раз – при вычислении *Pin* и *Pout*. В главном цикле вычисление *XN* (множества вершин с ненулевым потен­ци­а­лом) требует *O*(*n*+*m*) шагов, т. к. каждая вершина проходит через *STACK* один раз, а при удалении вершины удаляются и все инцидентные ей дуги. Каждая итерация главного цикла обращает в нуль потенциал по крайней мере одной вершины (*r*), т. е всего выполняется не более чем *n* итераций. Кроме вычисления *XN*, каждая итерация включает в себя ещё две части – процессы, подобные поиску в ширину – от *r* к *t* и от *s* к *r*. Сложность этих частей порядка числа анализируемых дуг. В процессе анализа дуга может быть удалена из сети (когда она насыщается потоком) – таких шагов *O*(*m*) – или сохранена для последующих проходов – таких дуг не более *n* (по одной на вершину). В сумме число шагов во всех итерациях главного цикла будет, таким образом, иметь порядок *m*+*n*2, т. е. *O*(*n*2). Следовательно, сложность всей функции *O*(*n*2).

В анализе сделано предположение, что удаление дуг из сети производится за время, ограниченное константой. Это возможно, если вхождения вершины *v* в *XLIST*[*u*] и вершины *u* в *XPREV*[*v*] взаимно связаны указателями и в оба списка содержат указатели и вперёд, и назад.

**Алгоритм построения максимального потока в сети  
 по методу Диница с помощью функций PSA и MAXPSA**

*Исходные данные*:

— *LIST*[*v*], *PREV*[*v*], *v* ∈ *V* – списки смежности и предшествования, задающие сеть;

— *С*[*u*][*v*], *u*, *v* ∈ *V* – пропускные способности дуг;

— *s* ∈ *V*, *t* ∈ *V* – источник и сток.

*Рабочая память*:

— *XV*, *F*[*u*][*v*] – множество вершин вспомогательной сети и поток в ней,

— *D*[*v*] – «расстояния» (к-ва дуг в увеличивающей цепи) от *s* до *v* во вспомогательной сети,

— *XLIST*[*u*][*v*], *XPREV*[*u*][*v*], *C*[*u*][*v*] – вспомогательная сеть.

*Результат*:

— *F*[*u*][*v*], *u*, *v* ∈ *V* – максимальный поток.

int main( ) {  
 for (auto u : V)   
 for(auto v :V) F[ u ][ v ] = 0;  
 do {  
 PSA( ); // построение вспомогательной бесконтурной сети  
 if ( D[t] != ∞) { // поток не является максимальным  
 MAXPSA( ); // построение максимального потока  
 // перенесение потока в главную сеть  
 for (auto u : XV)  
 for (auto v : XV) { // *XV* – множество в-н вспом. сети  
 F[ u ][ v ] += XF[ u ][ v ];  
 if (F[ u ][ v ] > C[ u ][ v ]) {  
 F[ v ][ u ] – = (F[ u ][ v ] – C[ u ][ v ]);  
 F[ u ][ v ] = C[ u ][ v ];  
 }  
 }  
 } // конец фазы  
 } while (D[t] != ∞); // поток *F* – максимальный  
}

По окончании работы алгоритма *PSA* *D*[*t*] = *l*. Если *l* = ∞, то сток *t* не достигнут, увеличивающей цепи нет, и поток в сети – максимальный. По достижении стока, когда *D*[*t*] < ∞, все остальные вершины уже не рассматриваются.

Далее во вспомогательной сети строится псевдомаксимальный поток *f\*.*  Это поток, для которого нельзя найти увеличивающей цепи длины *l* относительно этого потока, т. е. на некотором пути из *s* в *t* имеется дуга <*v*i, *v*i+1>, 0 ≤ *i* ≤ *l* такая, что *f\**(*vi, vi+*1) = *c*(*vi, vi*+1).

Далее поток *f\** переносится в основную сеть, складываясь с  *f* : *f\**(*u, v*) прибавляется к  *f*(*u, v*), а при превышении лимита *c* (*u, v*) производится уменьшение *f*(*v, u*). В результате получаем новый поток *W*( *f’* ) = *W*( *f* ) + *W*( *f\** ).

*Сложность*: исходя из анализа функций *PSA* и *MAXPSA*, а также учитывая, что число фаз не превосходит *n*, получаем для всего алгоритма оценку *O*(*n*3).

*Пример*. Вычисление максимального потока по алгоритму Диница



*Рис. 28.11*. Исходная сеть и первая фаза

Исходный поток полагаем нулевым. Ищем увеличивающую цепь. Вершины с нулевым потенциалом исключаем из обработки. Из оставшихся вершин ищем вершину с минимальным потенциалом. От неё, перемещая соответствующий «груз» в начало и в конец сети, находим псев­до­мак­си­маль­ный поток.



*Рис. 28.12*. Вторая фаза

Переносим найденный поток в исходную сеть и повторяем всё с начала. Обратите внимание на то, что поток, найденный в фазе 2 – не максимальный. Полученный результат в общем случае зависит от порядка, в котором обрабатывались вершины в множествах *XV*, *LIST*[*v*] и *PREV*[*v*]. После вершины 3 можно было начать перебор не с 7, а с 4. Результат был бы другим. А так понадобится ещё одна итерация.



*Рис. 28.13*. Третья фаза

Снова переносим найденный поток в исходную сеть. Между вершинами 7 и 3 насыщаем потоком согласованную дугу, а остаток реализуем умень­ше­ни­ем потока в несогласованной.



*Рис. 28.14*. Результат: максимальный поток

В полученной сети уже невозможно найти увеличивающую цепь, следовательно, найденный поток – максимальный.

*Замечание к реализации алгоритмов* *PSA* и *MAXPSA*. Представление графов тремя способами одновременно — матрицей пропускных способностей как обобщением матрицы смежности и наборами множеств смежности и предшествования представляется избыточным. Отказ от использования двух последних, не увеличивая оценку сложности, позволил бы упростить алгоритмы исключения рёбер из графа как результата обработки вершин с нулевым потенциалом.

*Контрольные вопросы*

1. Сколько фаз увеличения потока может потребоваться в алгоритме Диница для сети из 500 вершин и 2500 рёбер?
2. Является ли увеличивающая цепь самым коротким путём из источника в сток в сети, в которой ищется максимальный поток?
3. Является ли увеличивающая цепь самым коротким путём из источника в сток во вспомогательной бесконтурной сети, полученной первой фазой схемы Диница?

# 29. Паросочетания в двудольных графах

### 29.1. Основные понятия

***Паросочетание*** – это множество рёбер *M* неорграфа *G* = <*V, E*> такое, что никакие два ребра из *M* не инцидентны одной вершине:

∀ *e*1*, e*2 ∈ *M* ⇔ *e*1 = *e*2 ∨ (*e*1 ∩ *e*2 = ∅).

Вершина, не инцидентная ни одному ребру из *M* , называется ***свободной***.

Задача на отыскание паросочетания максимальной мощности *M* ⊆ *E* в неорграфе является *NP*-полной, т. е. решается полным перебором.

Интересны задачи на паросочетания в ***двудольных графах***, у которых множество вершин *V*  можно разбить на два непересекающихся подмно­же­ства *X* и *Y* таким образом, чтобы любое ребро графа соединяло вершину из *X* с некоторой вершиной из *Y*:

*V = X* ∪ *Y,* *X* ∩ *Y* = ∅; ∀ *e*∈ *E*, *e =* {*x, y*}, *x* ∈ *X, y* ∈ *Y*.

Введём для двудольного графа обозначение  *H =* <*X, Y, E*> – для некоторого фиксированного (не обязательно единственного) разбиения множества *V*. Задача на отыскание наибольшего (наибольшей мощности) паросочетания в двудольном графе известна как «задача о супружеских парах». Если интерпретировать  *X* как множество девушек, *Y –* как множество юношей, дуги из *E* – как взаимную симпатию, то требуется определить (сформировать) максимально возможное количество браков. Возможна и другая интер­пре­та­ция. Например, *X* – множество работников (станков, процессоров), *Y* – мно­жество работ (деталей, задач), дуга {*x*, *y*} означает, что работник *x* может выполнять работу (обрабатывать деталь, выполнять задачу) *y.* Цель отыскания наи­боль­шего паросочетания – найти оптимальную загрузку работников (станков, про­цес­со­ров), обеспечивающую наибольшую возможную производительность.

Эту задачу можно свести к отысканию максимального потока в некоторой сети.

Пусть задан двудольный граф *H =* <*X, Y, E*>. Построим сеть *S* ( *H* ) с источником *s* и стоком *t*

( *s* ≠ *t* ; *s*, *t* ∉ *X* ∪ *Y* ),

множеством вершин

*V\** = {*s*,*t*}∪ *X* ∪ *Y* ,

множеством дуг

*E\** = {< *s*, *x* > : *x* ∈ *X* } ∪ {< *y*, *t* > : *y* ∈ *Y* }  
 ∪ {< *x, y >*: *x* ∈ *X* ∧ *y* ∈ *Y* ∧ {*x, y*} ∈ *E* }

и пропускной способностью *c* ( *u*, *v*) = 1 для каждой дуги из *E\*.*

*Пример*.



*Рис*. *29.1*. Двудольный граф с паросочетанием и эквивалентная сеть с 0/1 потоком

*Теорема*. Существует взаимно однозначное соответствие между паросочетаниями в *H* и 0–1 потоками в *S* ( *H* ), при котором паросочетанию *M* = {{ *x*1, *y*1 },… { *x*k, *y*k }} (мощностью *k* ) соответствует поток величины *k* :

*f*M( *s*, *x*i ) = *f*M( *x*i , *y*i ) = *f*M( *y*i , *t*) = 1, 1 ≤ *i* ≤ *k*

и *f*M ( *e* ) = 0 для всех остальных дуг *e* сети *S* ( *H* ).

Потоку *f* величины *k* соответствует паросочетание

*Mf*  = {{ *x*, *y*} : *x* ∈ *X* ∧  *y* ∈ *Y* ∧  *f* ( *x*, *y*) = 1 }, | *Mf* | = *k*.

*Доказательство.* Все вершины в *M* попарно различны ⇔ функция *f*M является потоком.

*Следствие.* Для отыскания наибольшего паросочетания можно применить любой алгоритм определения максимального потока в сети, например, рассмотренную ранее схему Диница с временной сложностью *O* (*n*3).

Особая форма сети *S* ( *H* ) позволяет, однако, применить более простой алгоритм, предложенный Хопкрофтом и Карпом. Он основан на общей схеме Диница: по­сле­до­ва­тель­ность фаз увеличения потока вдоль увеличивающей цепи определённой длины.

Отметим, что каждая увеличивающая цепь в *S* ( *H* ) – нечётной длины и состоит (после отбрасывания первой и последней дуг) из некоторой последовательности чередующихся согласованных и несогласованных дуг, а первая и последняя дуга в этой цепи всегда согласованы.

Чередующаяся цепь (длины *l* = 2*k +* 1) из *X* в *Y* относительно  *M* – это множество дуг *P ⊆ E* вида *P* = {{*x*0, *y*1}, {*x*1, *y*2}, … {*y*k, *x*k}, {*x*k, *y*k+1}}, где *k*> 0, все *x*i, *y*i различны, *x*0 – свободная в *X*, *y*k+1 – свободная в *Y* и *каждая вторая* дуга ∈ *M*, т. е. *P* ∩ *M* = {{*y*1, *x*1}, {*y*2, *x*2}, … {*y*k, *x*k}}. Очевидно, что цепь представляет собой последовательность < *x*0, *y*1, *x*1, *y*2, … *y*k, *x*k, *x*k, *y*k+1>

При описанном в теореме соответствии между паросочетаниями в *H*  и нуль‑единичными потоками в *S* ( *H* ) чередующаяся цепь ⇔ увеличивающая цепь относительно потока *f*M в *S* ( *H* ), а именно цепь *s*, <*s*, *x*0>, *x*0, <*x*0, *y*1>, *y*1, <*y*1, *x*1>,*x*1, < *x*1, *y*2>,*y*2, … *y*k, <*y*k, *x*k>, *x*k, <*x*k, *y*k+1>, *y*k+1, < *y*k+1, *t* >, *t*, причём увеличение потока *f*M вдоль этой цепи даёт поток, соответствующий паросочетанию *M* ⊗ *P*.

*Пример*. Увеличение паросочетания *M* вдоль чередующейся цепи *P*.



*Рис*. *29.2.*Паросочетание, увеличивающая цепь и их объединение

*Теорема* [Берж]. Паросочетание *M* в двудольном графе *H* наибольшее ⇔ в H не существует чередующейся цепи относительно *M*.

### 29.2. Алгоритм Хопкрофта – Карпа для отыскания наибольшего паросочетания в двудольном графе

*Данные*: Двудольный граф *H =* <*X, Y, E*> в списках смежности  *LIST* [ *v* ], *v ∈ V*.

*Результат*: Наибольшее паросочетание в массиве  *COMB* [ *v* ], содержащем для каждой вершины  *v* номер вершины *u*, с которой сочетается *v*, или 0, если вершина *v* – свободная.

void PGA ( ) // Построение вспомогательного бесконтурного графа  
{ // *V*, *XV*, *X*, *Y*, *COMB* [ ], *LIST* [ ], *L* [ ], *XLIST* [ ], *s*, *t* – глобальные  
 for (u : V ∪ {s, t}) { L [ u ] = ∞ ; XLIST [ u ] = nullptr; } // Инициализация  
// Помещаем свободные  *x* ∈ *X* в очередь и список *XLIST* [ ]  
 queue <int> Q; XV = { s } ; L [ s ] = 0 ;  
 for (x : X) {  
 if ( COMB [ x ] == 0 ) { // *x* – свободная  
 Q.push( x );  
 XLIST [ s ] = ( XLIST [ s ] ∪ { x } ) ;//Добавляем *х* в сеть  
 L [ x ] = 1; //Расстояние от источника = 1  
 }  
 }

// Поиск в ширину, начиная со свободных вершин в *X*  
 while ( !Q.empty( ))  
 { u = Q.front( ); Q.pop( ); XV = ( XV ∪ { u } ) ;  
 if ( u ∈ Y) {  
 if ( COMB [ u ] = = 0 ) // (2) *u* – свободная, добавляем ребро <*u*, *t*>  
 { XLIST [ u ] = ( XLIST [ u ] ∪ { t } ) ;  
 XV = ( XV ∪ { t } ) ; L [ t ] = L[ u ] + 1 ;  
 }  
 else // *COMB* [ *u* ] != 0  
 { x = COMB [ u ] ;  
 if ( L [ x ] = = ∞ ) // Добавляем ребро <*u*, *x*>  
 { Q.push( x );   
 XLIST [ u ] = ( XLIST [ u ] ∪ { x } ) ; L [ x ] = L [ u ] + 1 ;  
}  
 }  
 }  
 else // *u* ∈ *X*  
 { for (y : LIST [ u ] ) {  
 if ( L [ u ] < L [ y ] ) // *L* [ *y* ] = *L* [ *u* ] + 1 или ∞  
 { if ( L [ u ] = = ∞ ) Q.push( y ); // *y* – новая  
 XLIST [ u ] = ( XLIST [ u ] ∪ { y } ) ; L [ y ] = L [ u ] + 1 ;  
 }  
 }  
 }// *else*  
 } // *while* (!*Q*.*empty*( ))  
 } // *PGA*

void PHASE ( ) // Увеличение паросочетания   
{ // Глобальные *XV*, *XLIST* [ ], *COMB* [ ], *s*, *t*; локальные *NEW* [ ], *P* [ ]  
 for ( u : XV ) // Инициализация  
 { NEW [ u ] = 1 ; P [ u ] = XLIST [ u ] ; }  
 // Поиск в глубину во вспомогательном графе от источника *s* stack <int> S ; S.push( s ); NEW [ s ] = 0 ;  
 while ( !S.empty( ) ) // Главный цикл  
 { u = S.top ( ) ; int b{0};   
 if ( P [ u ] = = nullptr ) b = 0;  
 else b = ! NEW [ P [ u ] –> d ] ;  
 while ( b ) {  
 P [ u ] = P [ u ] –> next;  
 if ( P [ u ] = = nullptr ) b = 0 ;  
 else b = ! NEW [ P [ u ] –> d ];  
 }  
 if ( P [ u ] ) // Найдена новая вершина  
 { if ( P [ u ] –> d = = t ) // Найдена чередующаяся цепь  
 while ( S.top ( ) ! = s )  
 { y = S.top( ); S.pop( ); x = S.top( ); S.pop( );   
 COMB [ x ] = y ; COMB [ y ] = x ;  
 } // В стеке осталась только вершина *s* else // P [ u ] –> d ! = t  
 { v = P [ u ] –> d ; S.push( v ); NEW [ v ] = 0 ; }  
 }  
 else // *P* [ *u* ] = = 0, т. е . в *XLIST* [ *u* ] нет новых вершин  
 S.pop( ); // Удаляем верхушку стека  
 } // Главный цикл  
} // *PHASE*

int main ( )

{ for ( v : V ) COMB [ v ] = 0; // Начинаем с пустого паросочетания

PGA ( );  
 do { PHASE ( ) ; PGA ( ) ; } while ( L [ t ] != ∞ ) ;  
}

Пояснения: *PGA* строит вспомогательный бесконтурный граф *H*M для двудольного графа *H* = < *X*, *Y*, *E* > и паросочетание *M* : ориентация рёбер *e*∈ *M* – от *Y* к *X*, а всех *e*∈ *E* \ *M* – от *X* к *Y*. Пути в *H*M от свободной вершины в *X* к свободной в *Y* в точности соответствуют чередующимся цепям относительно *M*.

*PGA* вначале строит дуги от *s* до всех свободных вершин *x*. Далее – поиск в ширину в графе *H*M от свободных вершин в  *X*. При нахождении свободной *y*∈ *Y* добавляем во вспомогательный граф дугу < *y*, *t*>.

*PHASE* – поиск в глубину во вспомогательном бесконтурном графе. При достижении стока  *t* содержимое стека – это чередующаяся цепь.  *t* не помещается в стек, и  *NEW* [ *t*] всегда 1, т. е. вершина  *t* может посещаться многократно. При установке *NEW* [*v* ] = 0 вершина *v* помещается в стек, и будет найдена проходящая через неё чередующаяся цепь или будет установлено, что такой цепи не существует.

*Пример*. Работа алгоритма Хопкрофта-Карпа



*Рис*. *29.3*. Исходный двудольный граф



*Рис*. *29.4*. Первая вспомогательная сеть (сток на расстоянии 3)



*Рис. 29.5.* Первое паросочетание



*Рис. 29.6*. Вторая вспомогательная сеть (сток на расстоянии 5)



*Рис*. *29.7.* Результат работы алгоритма: максимальное паросочетание.

Можно доказать, что число фаз алгоритма не превосходит *O* ( √ *n* ), что даёт общую оценку сложности *O* ((n + m) √ *n* ), или *O* ( *n*5/2).

Паросочетание *M* в двудольном графе *H* = <*X*, *Y*, *E*> назовём *совершенным*, если выполняется условие | *M* | = | *X* |, т. е. в паросочетание входят все вершины множества *X.*

**Условие существования** **совершенного** **паросочетания** (теорема Холла).

Совершенное паросочетание существует, если для любого *A* ⊆ *X* и множества вершин *B* ⊆ *Y,* смежных вершинам *A*  в двудольном графе *H*, выполняется условие | *A* | ≤ | *B* |.

Теорема Холла показывает, при каких условиях можно переженить всех юношей из множества *X* с девушками из множества *Y*. Для этого необходимо и достаточно, чтобы любые *k* юношей были знакомы в совокупности не менее чем с *k* девушками (1 ≤ *k* ≤ |  *X*  |).

## 29.3. Теория трансверсалей

Задачу на отыскание совершенного паросочетания можно рассматривать и в несколько иной интерпретации: можно рассмотреть семейство под­мно­жеств из *Y*, каждое из которых является множеством вершин, инци­дент­ных одной из вершин из *X.*

Если *Y* – произвольное конечное непустое множество и ℑ = ( *S*1, … *S*n ) – семейство непустых, но не обязательно различных его подмножеств, *трансверсалью* (или *системой различных представителей*) множества *Y* называется множество (мощностью *n* ) из элементов из подмножеств *S*i – по одному из каждого. Трансверсаль любого подмножества из ℑ называется *частичной трансверсалью* для *Y*.

**Условие существования трансверсали** (аналог теоремы Холла для трансверсалей): для указанных выше *Y* и ℑ трансверсаль существует ⇔ объединение любых *k* подмножеств из ℑ имеет мощность не менее *k*.

*Доказательство*: Необходимость очевидна.

Достаточность: если для некоторых *i* | *S*i | > 1, удалим по одному элементу из таких *S*i , не нарушив условия теоремы. Повторяя эту операцию многократно, придём к тому, что для всех *i* | *S*i | = 1, т. е. в *S*i останутся только представители.

Пусть заданы множество *Y* и набор его подмножеств ℑ , не обязательно не пересекающихся, не обязательно различных. Создадим множество *X* = {*x*1, …, *x*n}, все элементы которого попарно различны, и множество пар

*E* = {{*x*i, *y*j }: 1 ≤ *i* ≤ *n* ∧1 ≤ *j* ≤ *m* ∧ *y*j ∈ *S*i }.

Очевидно, что любая трансверсаль <*s*1, …, *s*n> однозначно соответствует паросочетанию мощности *n* в двудольном графе *H* = <*X*, *Y*, *E*>, а именно паросочетанию {{*x*1, *s*1}, …, {*x*n, *s*n}}.

Граф *H* имеет *n* + *m* вершин и  рёбер, следовательно, трансверсаль можно найти за время *O* (), используя алгоритм Хопкрофта – Карпа.

В рассмотренном выше примере можно считать, что задано множество *Y* мощностью 6 и набор его подмножеств ℑ = {{*y*1, *y*2, *y*3}, {*y*1, *y*2, *y*3, *y*4}, {*y*2, *y*3, *y*4, *y*5, *y*6}, {*y*1, *y*2, *y*3}, {*y*2, *y*3}, {*y*4}}. Трансверсаль, найденная алгоритмом Хопкрофта-Карпа: {*y*1, *y*2, *y*6, *y*4, *y*3}. Это частичная трансверсаль. Покажите, что для двудольного графа на рис. 29.3 условие существования полной трансверсали не выполняется.

## 29.4. Вершинное покрытие в двудольном графе

В произвольном графе *G* = <*V*, *E*> ***вершинное покрытие*** – это множество вершин *P* ⊆ *E* такое, что любое ребро *e* ∈ *E* инцидентно некоторой вершине *v* ∈ *P* .

*P* является вершинным покрытием ⇔ *N* = *V* \ *P* – ***независимое множество вершин***, т. е.

∀ *u*, *v* ∈ *V* \ *P* ⇒ {*u*, *v*} ∉ *E*

(никакие вершины из *N*  не связаны ребром).

Теорема. Пусть *M* – наибольшее паросочетание в двудольном графе  *H* = <*X*, *Y*, *E*> и пусть задано множество  *A* = {*a* ∈ *X* ∪ *Y*: *a* – вершина, достижимая из свободных вершин с помощью частичных чередующихся цепей} .

Если  *H* не содержит изолированных вершин, то *P* = (*X* \ *A*) ∪ (*Y* ∩ *A*) – вершинное покрытие, а *N* = (*X* ∩ *A*) ∪ (*Y* \ *A*) – независимое множество вершин.

Доказательство. Предположим, что некоторое ребро {*x*, *y*}∈ *E*, *x* ∈ *X*,  *y* ∈*Y* не инцидентно ни одной вершине из *P* ⇒ *x* ∈ *A*, *y* ∉ *A*.

{*x*, *y*} ∉ *M*, т. к. иначе вершина *x* была бы достижима с помощью цепи, проходящей через *y*, т. е. *y* ∈ *A.*

{*x*, *y*} ∉ *E* \ *M*, т. к. иначе цепь, достигающую  *x*, можно было бы увеличить на ребро {*x*, *y*} ⇒ *y* ∈ *A*. Таким образом,  *P* *является вершинным покрытием*.

Ни одна вершина в  *P*  *не свободна*.

Для *X* \ *A* это вытекает из определения *A*, для *Y* ∩ *A* – из того, что свободная вершина соответствовала бы чередующейся цепи относительно паро­со­че­та­ния *M*, которое по определению максимально.

*Не более одной вершины* каждого ребра {*x*, *y*} ∈ *M* принадлежит  *P*: *y* ∈ *Y* ∩ *A* ⇒ *x* ∈*A* ⇒ *x* ∉ *P* ⇒ | *P* | ≤ | *M* |.

Каждое вершинное покрытие должно содержать не менее одной вершины любого ребра. Это справедливо и для рёбер из паросочетания *M*, все вершины которого попарно различны ⇒ | *P* | ≥ | *M* |. Таким образом, | *P* | = | *M* |.

*N* – независимое множество вершин, т. к. *N* = (*X* ∪ *Y* ) \ *P*.

Множество  *A* можно найти за время *O* (| *X* | + | *Y*  | + | *E* |) обходом графа *H* в глубину или в ширину (как в алгоритме Хопкрофта – Карпа).

Следствие. В *двудольном* графе мощность минимального вершинного покрытия равна мощности максимального паросочетания.

Минимальное вершинное покрытие и максимальное независимое мно­жество вершин можно найти с помощью алгоритма Хопкрофта – Карпа за время *O* ((*m* + *n* ) ).

Если в графе *H* имеются изолированные вершины, их можно удалить из вершинного покрытия и добавить к независимому множеству вершин.

*Пример*: Минимальное вершинное покрытие как результат отыскания наибольшего паросочетания в двудольном графе.



*Рис*. *29.8*. Итоговая вспомогательная сеть: сток не достигнут.

В этом примере множество вершин вспомогательной сети *A* = {*x*1, *x*2, *x*4, *x*5, *x*6, *y*1, *y*2, *y*3, *y*4}. Следовательно, минимальное вершинное покрытие двудольного графа (рис. 13.3) *P* = {*x*3, *y*1, *y*2, *y*3, *y*4}, а максимальное независимое множество вершин *N* = {*x*1, *x*2, *x*4, *x*5, *x*6, *y*5, *y*6}.

Следует отметить, что для отыскания минимального вершинного покрытия в произвольном (недвудольном) графе алгоритмов полиномиальной сложности *не существует*.

*Контрольные вопросы*

1. Какой алгоритм целесообразно использовать для определения минимального вершинного покрытия в двудольном графе?
2. Какой алгоритм целесообразно использовать для определения минимального вершинного покрытия в произвольном неориентированном графе?
3. Сколько итераций может выполнить алгоритм Хопкрофта-Карпа для двудольного графа из 100 вершин и 2500 рёбер?
4. Какова мощность совершенного паросочетания в произвольном графе из 100 вершин и 2500 рёбер?
5. Какова временная сложность отыскания наибольшего паросочетания в произвольном неорграфе из n вершин и m рёбер?
6. Задано множество мощностью 50 и 100 его подмножеств суммарной мощностью 2500. Какова может быть максимальная мощность трансверсали?

# 30. Жадные алгоритмы решения оптимизационных задач

### 30.1. Понятие о матроиде

Алгоритм отыскания множества *S* = {*s*1, *s*2,… *sn*}, оптимального в некотором смысле, создаёт его, отбирая на каждом шаге элемент *si*, наилучший из доступных. Таким образом, глобальный оптимум достигается чередой выбора локальных оптимумов. Алгоритм, действующий таким образом, называется жадным. Примером такого алгоритма является рассмотренный в своё время алгоритм Дейкстры для отыскания вектора расстояний в графе с на­гру­женными рёбрами.

Мы рассмотрим условия, при которых такая стратегия действительно может дать желаемый результат за полиномиальное время.

Рассмотрим матрицу с вещественными неотрицательными коэффициентами.

7 5 1  
 3 4 3  
 2 3 1

*Задача* 1. Найти подмножество элементов матрицы с наибольшей суммой. Ограничение: разрешается брать по одному элементу в столбце.

Алгоритм: просматриваем столбцы, берём наибольший из каждого. Результат:

**7** **5** 1  
 3 4 **3**  
 2 3 1

Сумма весов 7+5+3=15.

*Задача* 2. Найти подмножество элементов матрицы с наибольшей суммой. Ограничение: разрешается брать по одному элементу в столбце и в строке.

Алгоритм: просматриваем столбцы, берём наибольший из каждого, пропускаем уже использованные строки. Результат:

**7** 5 1  
 3 **4** 3  
 2 3 **1**

Сумма весов 7+4+1=12. Правильный результат:

**7** 5 1  
 3 4 **3**  
 2 **3** 1

Сумма весов 7+3+3=13.

*Задача 3*. Дано конечное множество *E,* семейство его подмножеств ℑ ⊆ *P* (*E*) и функция w : *E* → *ℝ*+, где *ℝ*+ — множество вещественных неотрицательных чисел. Найти подмножество *S* ∈ ℑ с наибольшей суммой *w* (*e*), *e* ∈ *S*.

Для того, чтобы жадный алгоритм правильно решал задачу для про­из­воль­ной функции *w*, достаточно, чтобы пара < *E, ℑ* > образовывала *матроид*, т. е. множество и набор *линейно независимых* его подмножеств.

Семейство ℑ ⊆ *P* (*E*) при этом удовлетворяет условиям:

**М1**: ∅ ∈ ℑ и если *A* ∈ ℑ и *B* ∈ *A*, то *B* ∈ ℑ.

**М2**: Для произвольных *A, B* ∈ ℑ и *B* ⊆ *A* таких, что *\A\* = *\B\* + 1, существует элемент *e*∈ *B* \ *A*, такой, что *A* ⋃ {*e*} ∈ ℑ.

Множества ℑ — *независимые*, а множества *P* (*E*) \ ℑ — *зависимые множества* матроида М. Подмножество {*e*1, … , *en*} линейного пространства называется независимым, если не существует набора скаляров λ1, … , λ*n* не всех равных нулю такого, что λ1 *e*1 + … λ*n en* = 0.

*C* ⊆ *E*  — *максимальное независимое* подмножество *E*, если не су­щест­вует независимого подмножества *B* такого, что *C ⊂ B* ⊆ *E.*

Для матроида *M*  выполняется условие

**М3**: Любые максимальные независимые подмножества  *C* ⊆ *E*  имеют одинаковую мощность.

М1 и М3 образуют эквивалентную М1 и М2 систему аксиом для мат­ро­идов.

Максимальное независимое подмножество  *C* ⊆ *E —* это *база* матроида, мощность такого подмножества — это *ранг r*(*E*) множества *E*, ранг матроида, аналог размерности линейного пространства. Каждое независимое подмножество *B* ⊆ *E* можно расширить до базы добавлением элементов из *E.*

Примеры матроидов

1. Свободный матроид < *E, P*(*E*) >. В таком матроиде каждое множество *A* ⊆ *E* независимое, следовательно, *r*(*A*) = | *A* |.
2. Матроид разбиений (*см*. задачу 1). Для разбиения *π* = {*D*1, … , *Dk*} множества *E*

ℑ = { *A* ⊆ *E* : |*A* ⋂ *Di* | ≤ 1 для *i* = 1, … , *k*}.

### 30.2. Жадный алгоритм для матроидов (теорема Рамо — Эдмонса)

Для получения независимого множества *S* с наибольшим весом (задача 3) пригоден следующий алгоритм (общая схема жадного алгоритма).

1. Упорядочить множество *E*  по убыванию весов так, чтобы

*E* = {*e*1, … , *en*}, *w*(*e*1) ≥ *w*(*e*2) ≥ … ≥ *w*(*en*).

1. *S*  = ∅.
2. for (i = 1; i ≤ n; ++i)

if (*S* ⋃ { *ei* } ∈ ℑ ) *S*  = (*S* ⋃ { *ei* }).

Множество *S,* полученное жадным алгоритмом, оптимально в очень сильном смысле: у него не только максимальна сумма весов, она мак­си­ма­льна и для любого его независимого подмножества (промежуточного результата). В матроиде нельзя выбрать независимое множество, состоящее из меньшего числа больших по весу элементов.

После упорядочения исходного множества *E* веса его элементов нас больше не интересуют.

### 30.3. Матричные матроиды: независимое множество векторов

Пусть {*v*1, … , *vm*} — элементы некоторого линейного пространства размерности *n* с базисом {*b*1, … ,*bn*}. Векторы *v*1, … , *vm*имеют однозначное разложение относительно базиса *b*1, … ,*bn.*

***v*1 = *a*11*b*1 + *a*21*b*2 + … + *an*1*bn***

***v*2 = *a*12*b*1 + *a*22*b*2 + … + *an*2*bn***

***vn*= *a*1*nb*1 + *a*2*nb*2 + … + *anmbn***

Рассмотрим матрицу

***a*11 *a*12  … *a*1*m***

***a*21 *a*22  … *a*2*m***

***A* = . . .**

***an*1 *an*2  … *anm***

Её столбцы *e*1, … *em* соответствуют векторам *v*1, … , *vm*, причём множество векторов линейно независимо тогда и только тогда, когда линейно независимо соответствующее им множество столбцов. И обратно, столбцы произвольной матрицы *A* указанного выше вида можно трактовать как векторы некоторого *n-*мерного линейного пространства. Каждая такая матрица определяет матроид *M* (*A*) = < *E*, ℑ >, где *E* есть множество её столбцов, а *B* ∈ ℑ тогда и только тогда, когда множество столбцов *B* линейно независимо. Матроид, определённый таким образом, будем называть *матричным*.

***Жадный алгоритм для матричного матроида***

*Дано*: матрица *A* из *n* строк и *m* столбцов со столбцами, упорядоченными по не возрастанию весов (неотрицательных).

*Результат*: независимое множество *S* номеров столбцов с наибольшей суммой весов.

*S*  = ∅; *j* = 0;

for (int *i* = 0; *i* < *n*; ++*i*)

{

while ((*j* < *m*) && (abs(*A*[ *i* ][ *j* ]) < eps)) ++*j*;

if((*j* < *m*)&&abs(*A*[ *i* ][ *j* ])>eps) // Столбец *j* не нулевой

{ *S* = (*S* ⋃ { *j* });

for (int *k* = *j* +1; *k* < *m*; ++*k*){

*Aijk* = *A*[*i*][*k*]/*A*[*i*][*j*];

for (int *i* = 0; *i* < *n*; ++*i*) { *A*[ *i* ][ *k* ] –= *A*[ *i*][ *j* ] \* *Aijk*;

}

if(*j* => *m*)break;

}

Процесс преобразования матрицы *A* этим алгоритмом –– это известный из численного анализа метод исключения Гаусса. В главном цикле сначала проверяется, не состоит ли столбец *j* из одних нулей. Если да, то он не принадлежит к линейно независимому множеству столбцов. Если нет, то мы включаем номер столбца *j* в решение *S* и пересчитываем все оставшиеся столбцы. Это не нарушает линейной независимости столбцов, но обнуляет все элементы *i*-той строки справа от *A*[ *i* ][ *j* ]. По окончании работы *S* содержит номера ненулевых столбцов. Эти столбцы линейно независимы: после соответствующей перестановки строк они содержат матрицу размером |*S*| × |*S*|с ненулевой главной диагональю и с нулями выше её.

Сложность алгоритма легко оценить, она составляет *O*(*n*2*m*).

*Пример*. Результат обработки алгоритмом Гаусса матрицы из 10 строк и 15 столбцов (векторов, упорядоченных слева направо по не возрастанию веса). Вектор 2 представляет собой удвоенную копию вектора 3, а вектор 6 – сумму векторов 4 и 5.

**Source 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 1.7 0 0 1.9 2.4 4.3 0.8 1.2 1.4 0.5 4.5 3.1 2.7 1.1**

**1: 4.1 4.5 5.4 2.7 3.6 4.1 7.7 0.2 0.3 4.2 3.2 2.1 1.6 1.8 4.5**

**2: 4.7 2.6 7.6 3.8 1.9 1.2 3.1 4.9 3.5 4.4 0.3 1.1 2.2 3.3 2.3**

**3: 1.4 4.1 0.6 0.3 1.8 4.7 6.5 1.2 0.7 3.7 0.9 2.3 4.1 2.9 2.8**

**4: 1.6 3.5 8.4 4.2 3.8 0.6 4.4 4.2 1.4 4.8 4.6 0.5 4 2.9 2**

**5: 0 0.6 8.6 4.3 4.8 2.9 7.7 3.4 0.4 0.6 4 1.6 2.6 3.1 0.8**

**6: 4.4 3.9 4.6 2.3 3.7 3.8 7.5 3.2 2.9 4.1 3.3 1.5 3.9 0.8 0.4**

**7: 3 2.7 4.6 2.3 3.6 2.1 5.7 2.4 2.2 2 2.9 2.7 2.3 4.7 1.2**

**8: 3.6 4 7.2 3.6 0.5 1.7 2.2 2.4 3.1 0.2 0 0 4.1 2.4 1.6**

**9: 3 0.7 1.4 0.7 3.7 0.7 4.4 0.3 3.3 4.5 0.9 0.9 0.8 2.1 3.8**

*Рис*.*30.1*. Исходный набор векторов

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 5.4 2.7 1.7 1.7 3.4 -0.6 -0.9 2.8 2.7 -2.4 -1.5 -0.9 3.4**

**2: 4.7 0.65 7.6 3.8-0.28 -1.6 -1.8 4 2.1 2.8-0.27 -4.1 -1.4 0.2 1**

**3: 1.4 3.5 0.6 0.3 1.2 3.9 5 0.93 0.29 3.2 0.73 0.76 3 2 2.4**

**4: 1.6 2.8 8.4 4.2 3.1-0.34 2.7 3.9 0.93 4.3 4.4 -1.3 2.8 1.8 1.6**

**5: 0 0.6 8.6 4.3 4.8 2.9 7.7 3.4 0.4 0.6 4 1.6 2.6 3.1 0.8**

**6: 4.4 2.1 4.6 2.3 1.7 1.2 2.9 2.3 1.6 2.6 2.8 -3.3 0.57 -2.1-0.78**

**7: 3 1.5 4.6 2.3 2.2 0.34 2.6 1.8 1.3 0.98 2.5-0.590.032 2.7 0.4**

**8: 3.6 2.5 7.2 3.6 -1.2-0.41 -1.6 1.7 2 -1-0.44 -4 1.40.029 0.63**

**9: 3-0.54 1.4 0.7 2.3 -1.1 1.3-0.29 2.4 3.5 0.53 -2.4 -1.5 0.12 3**

*Рис*.*30.2*. Шаг 0

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 3.2-0.67 -1.9 -2.6 4.1 2.3 2.1 -0.9 -3.5 -1 0.41 0.25**

**3: 1.4 3.5 -6.2 -3.1-0.99 1.7 0.76 1.7 1.4 -0.3 -2.7 3.8 4.9 3.1 -1.8**

**4: 1.6 2.8 2.9 1.5 1.3 -2.1 -0.72 4.5 1.8 1.4 1.7 1.2 4.3 2.8 -1.9**

**5: 0 0.6 7.4 3.7 4.4 2.5 7 3.5 0.59 0 3.4 2.1 2.9 3.3 0.071**

**6: 4.4 2.1 0.6 0.3 0.4-0.036 0.36 2.8 2.3 0.52 0.76 -1.6 1.7 -1.4 -3.3**

**7: 3 1.5 1.8 0.9 1.3-0.54 0.79 2.1 1.8-0.48 1.1 0.66 0.81 3.2 -1.4**

**8: 3.6 2.5 2.4 1.2 -2.7 -1.9 -4.6 2.2 2.9 -3.5 -2.9 -1.8 2.7 0.84 -2.4**

**9: 3-0.54 2.4 1.2 2.6-0.73 1.9 -0.4 2.2 4 1.1 -2.9 -1.8-0.05 3.7**

*Рис*. *30.3*. Шаг 1

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 -0.15 -1.8 5.7 3.7 1.8 -3.5 0.37 3.9 3.5 -1.6**

**4: 1.6 2.8 2.9 0 1.6 -1.2 0.49 2.6 0.77 0.43 2.1 2.8 4.8 2.6 -2**

**5: 0 0.6 7.4 0 5.2 4.8 10 -1.3 -2.1 -2.5 4.5 6.2 4.1 2.8 -0.22**

**6: 4.4 2.1 0.6 0 0.46 0.15 0.61 2.4 2.1 0.32 0.85 -1.2 1.8 -1.5 -3.3**

**7: 3 1.5 1.8 0 1.5 0.096 1.5 0.96 1.1 -1.1 1.4 1.6 1.1 3.1 -1.4**

**8: 3.6 2.5 2.4 0 -2.4 -1.2 -3.6 0.7 2 -4.3 -2.5 -0.5 3.1 0.68 -2.5**

**9: 3-0.54 2.4 0 2.9 0.026 2.9 -2 1.3 3.2 1.4 -1.5 -1.4 -0.21 3.6**

*Рис*.*30.4.* Шаг 2

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 -1.3 8.3 4.5 2.2 -1.5 3.2 8.7 6.1 -3.6**

**5: 0 0.6 7.4 0 5.2 4.3 4.3 17 9.6 3.2 -6.8 7.4 17 14 -5.3**

**6: 4.4 2.1 0.6 0 0.46 0.1 0.1 4 3.1 0.83-0.15 -1.1 2.9-0.48 -3.8**

**7: 3 1.5 1.8 0 1.5-0.13-0.13 6.2 4.5 0.57 -1.9 2 4.7 6.3 -2.9**

**8: 3.6 2.5 2.4 0 -2.4-0.97-0.97 -7.8 -3.5 -7 2.7 -1 -2.8 -4.5-0.12**

**9: 3-0.54 2.4 0 2.9-0.25-0.25 8.1 7.9 6.4 -4.8-0.86 5.6 6 0.72**

*Рис*.*30.5*. Шаг 3 (вектор 3 пропущен)

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 24 11 -12 18 45 34 -17**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 3.5 1-0.27-0.87 3.60.0032 -4.1**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 4.1 0.34 -1.7 1.7 3.9 5.7 -2.6**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 -6.8 -8.6 3.8 -3.4 -9.2 -9 2.5**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 7 5.9 -4.6 -1.5 4 4.8 1.4**

*Рис*.*30.6*. Шаг 4

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 0 0 0 0 0 0 0**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 0.89-0.11 0.96 -2.7 -1.2 -3.6 -2.2**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 1.1-0.94-0.32 -0.5 -1.7 1.6-0.47**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 0.85 -5.3 0.16 2.2 5.1 1.7 -2.9**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 3.4 4.4 -2.9 -4.1 -2.7-0.17 3.9**

*Рис*.*30.7*. Шаг 5 (вектор 6 пропущен)

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 0 0 0 0 0 0 0**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 0.89 0 0 0 0 0 0**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 1.1 -0.8 -1.5 3-0.14 6.1 2.4**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 0.85 -5.2-0.75 4.9 6.2 5.1-0.71**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 3.4 4.8 -6.6 6.6 2 14 13**

*Рис*.*30.8.* Шаг 6

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 0 0 0 0 0 0 0**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 0.89 0 0 0 0 0 0**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 1.1 -0.8 0 0 0 0 0**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 0.85 -5.2 9.3 -15 7.2 -35 -16**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 3.4 4.8 -16 24 1.1 50 27**

*Рис*.*30.9*. Шаг 7

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 0 0 0 0 0 0 0**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 0.89 0 0 0 0 0 0**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 1.1 -0.8 0 0 0 0 0**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 0.85 -5.2 9.3 0 0 0 0**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 3.4 4.8 -16-0.39 13 -8.8-0.75**

*Рис*.*30.10*. Шаг 8

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

**0: 4.1 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**1: 4.1 2.8 0 0 0 0 0 0 0 0 0 0 0 0 0**

**2: 4.7 0.65 6.3 0 0 0 0 0 0 0 0 0 0 0 0**

**3: 1.4 3.5 -6.2 0 -1.6 0 0 0 0 0 0 0 0 0 0**

**4: 1.6 2.8 2.9 0 1.6 -1.3 0 0 0 0 0 0 0 0 0**

**5: 0 0.6 7.4 0 5.2 4.3 0 44 0 0 0 0 0 0 0**

**6: 4.4 2.1 0.6 0 0.46 0.1 0 4.7 0.89 0 0 0 0 0 0**

**7: 3 1.5 1.8 0 1.5-0.13 0 5.4 1.1 -0.8 0 0 0 0 0**

**8: 3.6 2.5 2.4 0 -2.4-0.97 0 -14 0.85 -5.2 9.3 0 0 0 0**

**9: 3-0.54 2.4 0 2.9-0.25 0 6.5 3.4 4.8 -16-0.39 0 0 0**

*Рис*.*30.11*. Шаг 9

*Результат*: отобраны вектора <0 1 2 4 5 7 8 9 10 11>.

### 30.4. Графовые матроиды: стягивающее дерево наименьшей стоимости

Пусть *G* = <*V*, *E*> — неориентированный граф.

Определим *M* (*G*) = < *E*, ℑ >, где ℑ = {*A* ⊆ *E* : граф <*V*, *A*> не содержит циклов}. *M* (*G*) является *матроидом для произвольного графа* *G.* Это так, поскольку каждое максимальное подмножество *A* ∈ ℑ, содержащееся в множестве *B* ⊆ *E*, является стягивающим лесом графа <*V*, *B*>.

Графовый матроид *M* (*G*) можно трактовать как матричный матроид, соответствующий матрице инциденций графа *G*, рассматриваемой как матрица с элементами из двухэлементного поля *Z*2 = {0, 1}. сложение в таком поле выполняется по модулю 2, а умножение — как обычно для целых чисел. Строки матрицы инциденций соответствуют вершинам, а столбцы — рёбрам. Подмножество столбцов матрицы линейно зависимо, если соответствующий ему подграф содержит цикл.

Если каждому столбцу матрицы инциденций приписать вес, алгоритм Гаусса позволит получить независимое подмножество столбцов с наименьшей суммой весов — стягивающий лес наименьшей стоимости — за время *O* (*n*2*m*). Но, к сожалению, Алгоритм Гаусса здесь неприменим, поскольку линейная комбинация векторов — столбцов матрицы инциденций имеет совсем другой смысл.

Линейная комбинация столбцов превращается в вектор из нулей, если входящие в неё рёбра образуют цикл. Рассмотрим в качестве примера комбинацию из рёбер {*a*,*b*}, {*a*,*c*}, {*b*,*c*}. В этом наборе каждая вершина упоминается ровно 2 раза, следовательно, симметрическая разность соответствующих столбцов будет состоять из одних нулей. Но после присоединения к ней других рёбер это свойство исчезает и выявить цикл становится невозможно. Следовательно, для обнаружения цикла следует отдельно проверить каждую тройку, четвёрку и т. д., которые могут получиться из уже отобранных рёбер в результате добавления в решение ещё одного ребра.

Это можно сделать следующим образом.

1. Поместить в решение нулевой столбец матрицы (с минимальным весом):

*S* = (*S* ⋃ { 0});

Номер добавляемого столбца *j* установить равным 1, позицию начала набора комбинаций для добавляемого ребра *p* и позицию для очередной комбинации рёбер *q* установить равными *m*: комбинации будут располагаться в матрице инциденций правее рёбер.

1. Добавить в столбец *q* матрицы комбинацию ребра *j* поочередно со всеми рёбрами, уже отобранными в *S*, каждый раз увеличивая *q* на 1. Эти комбинации не будут нулевыми, потому что пары рёбер не образуют цикла. В первый раз это будет единственная комбинация из рёбер 0 и 1.
2. Этот шаг выполняется только для *j* > 1. Продолжаем формировать комбинации, добавляя ребро *j* к каждой имеющейся в позициях от *m* до *p*. Эти комбинации будут соответствовать тройкам, четвёркам и т. д. рёбер, и нулевое количество единиц в любой из них будет означать, что ребро *j* замыкает цикл. Если такая комбинация обнаружится, ребро j пропускается, а все комбинации с его участием сбрасываются (*q* = *p*).  
   Если нулевой комбинации не найдено, ребро *j* добавляется в решение, и комбинации с его участием закрепляются (*p* = *q*).

*S =* (*S* ⋃{ *j* })*;*

1. Переход к следующему ребру: *j* = *j* + 1 и возврат к шагу 2. Цикл заканчивается, когда в решение будут отобраны (n – 1) рёбер (получено стягивающее дерево) или использованы все имеющиеся рёбра (получен стягивающий лес).



*Рис*.*30.12*. Граф с нагруженными рёбрами   
и его стягивающее дерево наименьшей стоимости

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | 1 | 1 |  |  |  | 1 |  |  |  |  |  |  |  |
| b | 1 |  |  |  |  |  | 1 |  |  |  |  | 1 |  |
| c |  | 1 | 1 |  |  |  | 1 | 1 |  |  |  |  | 1 |
| d |  |  |  |  | 1 |  |  |  | 1 |  |  |  | 1 |
| e |  |  |  | 1 | 1 |  |  |  |  |  |  | 1 |  |
| f |  |  |  |  |  | 1 |  | 1 |  |  | 1 |  |  |
| g |  |  | 1 |  |  |  |  |  | 1 | 1 | 1 |  |  |
| h |  |  |  | 1 |  |  |  |  |  | 1 |  |  |  |
| вес: | **1.5** | **3** | **3.7** | **4** | **5** | **6** | 7 | 8 | **8.3** | 9 | 11 | 12 | 14 |

*Рис*. *30.13*. Матрица инциденций, упорядоченная по весам столбцов

*Пример*. Результат прогона программы испытания графового матроида.

**n=8 m=13**

**Source 0 1 2 3 4 5 6 7 8 9 10 11 12**

**a: 1 1 0 0 0 1 0 0 0 0 0 0 0**

**b: 1 0 0 0 0 0 1 0 0 0 0 1 0**

**c: 0 1 1 0 0 0 1 1 0 0 0 0 1**

**d: 0 0 0 0 1 0 0 0 1 0 0 0 1**

**e: 0 0 0 1 1 0 0 0 0 0 0 1 0**

**f: 0 0 0 0 0 1 0 1 0 0 1 0 0**

**g: 0 0 1 0 0 0 0 0 1 1 1 0 0**

**h: 0 0 0 1 0 0 0 0 0 1 0 0 0**

*Рис*.*30.14.* Исходная матрица инциденций (упорядочена).   
Рёбра (столбцы матрицы) пронумерованы от (0) до (12)

**First edge {a,b}**

**Step 2 p=13 q=14 Edge {a,c} Added!**

**13**

**0: 0**

**1: 1**

**2: 1**

**3: 0**

**4: 0**

**5: 0**

**6: 0**

**7: 0**

*Рис*.*30.15.* Шаг 1. Ребро (0) отобрано безусловно. Шаг 2. Добавлено ребро (1).  
В добавленном столбце *m* — их линейная комбинация (0)⊕(1)

**Step 3 p=14 q=17 Edge {c,g} Added!**

**14 15 16**

**0: 1 1 0**

**1: 1 0 1**

**2: 1 0 0**

**3: 0 0 0**

**4: 0 0 0**

**5: 0 0 0**

**6: 1 1 1**

**7: 0 0 0**

*Рис*. *30.16.* Шаг 3. Добавлено ребро (2). В добавленных столбцах линейная комбинация (0)⊕(1)⊕(2) — ненулевая, и новые пары (0)⊕(2) и (1)⊕(2).  
Значения *p* и *q* показывают интервал столбцов [14 … 17),  
в формировании которых участвует ребро (2)

**Step 4 p=17 q=24 Edge {e,h} Added!**

**17 18 19 20 21 22 23**

**0: 1 1 0 0 1 1 0**

**1: 1 0 0 1 1 0 1**

**2: 0 1 1 1 1 0 0**

**3: 0 0 0 0 0 0 0**

**4: 1 1 1 1 1 1 1**

**5: 0 0 0 0 0 0 0**

**6: 0 0 1 0 1 1 1**

**7: 1 1 1 1 1 1 1**

*Рис*. *30.17*. Шаг 4. Добавлено ребро (3).  
В добавленных столбцах линейные комбинации из всех четвёрки и трёх троек из отобранных рёбер — все ненулевые, и новые пары из ребра (3)  
с отобранными ранее (0), (1) и (2)

**Step 5 p=24 q=39 Edge {d,e} Added!**

**24 25 26 27 28 29 30 31 32 33 34 35 36 37 38**

**0: 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0**

**1: 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1**

**2: 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0**

**3: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

**4: 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0**

**5: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**6: 0 0 1 0 0 1 1 1 0 0 1 0 1 1 1**

**7: 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1**

*Рис*. *30.18*. Шаг 5. Добавлено ребро (4).  
В столбцах [24, 39) — все линейные комбинации с его участием (ненулевые)

**Step 6 p=39 q=70 Edge {a,f} Added!**

**39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69**

**0: 0 0 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 1**

**1: 1 0 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1**

**2: 0 1 1 0 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0**

**3: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

**4: 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0**

**5: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

**6: 0 0 1 0 0 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 1 1**

**7: 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1**

*Рис*. *30.19*. Шаг 6. Добавлено ребро (5). Добавляемых комбинаций всё больше

**Step 7 p=70 q=70 Edge {b,c} Skipped!**

**70 71 72 73 74 75 76**

**0: 1 1 0 0 0 1 0**

**1: 0 1 1 1 1 1 0**

**2: 1 0 0 1 1 1 0**

**3: 0 0 0 0 1 0 0**

**4: 0 0 0 1 1 0 0**

**5: 0 0 0 0 0 1 0**

**6: 0 0 1 0 0 0 0**

**7: 0 0 0 1 0 0 0**

*Рис. 30.20*. Шаг 7. Ребро (6) пропускается: среди добавленных комбинаций нашлась нулевая — из рёбер, образующих цикл. Это комбинация 76: ((0)⊕(1)⊕(6)). Формирование комбинаций прервано. Столбцы с участием ребра (6) исключаются из обработки (*p* = *q* = 70)

**Step 7 p=70 q=70 Edge {c,f} Skipped!**

**70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103**

**0: 1 1 0 0 0 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 0 0**

**1: 1 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0**

**2: 1 0 0 1 1 1 0 0 1 1 1 0 0 0 0 1 1 1 0 0 1 0 0 1 1 1 0 0 0 0 1 1 1 0**

**3: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0**

**4: 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0**

**5: 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0**

**6: 0 0 1 0 0 0 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 0**

**7: 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 0**

*Рис*. *30.21*. Повторение шага 7 для ребра (7).   
Оно тоже пропускается: нулевой оказалась комбинация 103: ((1)⊕(5)⊕(7))

**Step 7 p=70 q=133 Edge {d,g} Added!**

**70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132**

**0: 1 1 0 0 0 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 0 0 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 1**

**1: 1 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1**

**2: 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 0 0**

**3: 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**4: 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0**

**5: 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

**6: 1 1 0 1 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 0 0 0**

**7: 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1**

**Result: {a,b} {a,c} {c,g} {e,h} {d,e} {a,f} {d,g}**

*Рис*. *30.22.* Ещё одно повторение шага 7 — для ребра (8).   
Нулевых комбинаций не оказалось, и ребро добавляется в решение.   
Поскольку отобрано (*n*–1) = 7 рёбер, они образуют результат —   
искомое стягивающее дерево наименьшей стоимости

Как видно из демонстрации работы алгоритма, его сложность определяется количеством комбинаций из уже отобранных рёбер и временем для их проверки, т. е. проверка добавляемого ребра на линейную независимость с уже отобранными требует O(*n*.*n*!) шагов, что даёт для алгоритма общую оценку временной сложности O(*m*.*n.n*!) и ёмкостной сложности O(*n.n*!) — из-за расхода памяти для хранения линейных комбинаций рёбер.

Если реализовать столбцы матрицы и комбинации из них машинными словами, сложность получится O(*m.n*!), т. е. всё равно останется неприемлемой.

Однако для задачи отыскания стягивающего дерева наименьшей стоимости существуют и более быстрые алгоритмы.

# 31. Эффективные алгоритмы для отыскания стягивающего дерева наименьшей стоимости и матроиды трансверсалей

### 31.1. Алгоритм Краскала

*Дано*: неорграф *G* = <*V*, *E*> с функцией *E* → *ℝ*+ задан в виде вектора из *m* пар *ei* = {*u*, *v*}, упорядоченных по не возрастанию весов.

*Результат*: независимое множество *T* номеров столбцов с наименьшей суммой весов — стягивающее дерево <*V*, *T*>.

*T*  = ∅;  
*UF*  = {{*v*} : *v* ∈ *V*};  
for (int *i* = 0; *i* < *m*; ++*i*)  
{ if( *ei* = {*u*, *v*}, где *u* ∈ *A*, *v* ∈ *B*, *A,B* ∈ *UF*, *A!=B*)  
 { *T =* (*T* ⋃ {*ei*});  
 *UF =* (*UF* \ {*A, B*}) ⋃ {*A* ⋃ *B*};  
 }  
}

Сложность *O*(*m* log *m*) = *O*(*m* log *n*) — определяется сортировкой. Рекомендуется для редких графов.

*Пример*. Граф задан очередью из рёбер, упорядоченных по неубыванию веса (рис.31.1): <{*a*,*b*}, {*a*,*c*}, {*c*,*g*}, {*e*,*h*}, {*d*,*e*}, {*a*,*f*}, {*b*,*c*}, {*c*,*e*}, {*d*,*g*}, {*g*,*h*}, {*f*,*g*}, {*b*,*e*}, {*c*,*d*}. *UF* = <1, 2, 3, 4, 5, 6, 7, 8>.



*Рис*. *31.1*. Первый шаг алгоритма Краскала. *UF* = <1, 1, 3, 4, 5, 6, 7, 8>.



*Рис*.*31.2*. Второй шаг. *UF* = <1, 1, 1, 4, 5, 6, 7, 8>.



*Рис*.*31.3*. Третий шаг. *UF* = <1, 1, 1, 4, 5, 6, 1, 8>.



*Рис*.*31.4*. Четвёртый шаг. *UF* = <1, 1, 1, 4, 5, 6, 1, 5>.



*Рис*.*31.5.* Пятый шаг. *UF* = <1, 1, 1, 4, 4, 6, 1, 4>.



*Рис*.*31.6*. Шестой шаг. *UF* = <1, 1, 1, 4, 4, 1, 1, 4>.



*Рис*.*31.7*. Седьмой шаг (результат). *UF* = <1, 1, 1, 1, 1, 1, 1, 1>.

### 31.2. Алгоритм Прима

Этот алгоритм наиболее прост в реализации среди всех алгоритмов поиска стягивающего дерева наименьшей стоимости (MST) и рекомендуется для насыщенных графов. В нём используются рёбра дерева mst, самое короткое ребро, соединяющее дерево с остальными вершинами и его длина.

Задан неорграф *G* = <*V*, *E*> и функция *E* → *ℝ*+. Требуется найти максимальный подграф <V, T>, T ⊆ E, не содержащий циклов, с наименьшей суммой весов образующих его рёбер.

T = { v0 } //Начинаем с любой вершины  
while( V\T ≠ ∅ )  
{ (Выбрать ребро <u, v> наименьшей стоимости: u ∈ T, v ∈ E\T);  
 T = T ∪ { v };  
}

Если представить множество рёбер матрицей весов, сложность алгоритма *O*(*n*3). Если для отыскания ребра с наименьшим весом применить пирамиду, сложность будет *O*(*m* log *n*). Рёбра, соединяющие вершины из *T* с вершинами из *V*\*T*, образуют так называемую «кайму». Их можно поместить в контейнер *priority\_queue*. Начальное содержимое контейнера определяется множеством смежности стартовой вершины *v*0. В дальнейшем из очереди удаляются как очередное ребро с наименьшим весом, так и рёбра. замыкающие циклы, и добавляются рёбра, инцидентные вновь присоединяемой вершине.

Пример реализации:

class EDGE { //ребро  
 public:  
 EDGE(int, int, double);  
 int v( )const; int w( ) const;  
 double wt( ) const;  
 bool from(int) const;  
 int other(int) const;  
 void show( ) {cout ≪ v( ) ≪ '-' ≪ w( ); }  
};  
template <class Edge>   
class GRAPH {  
 public:  
 GRAPH(int, bool);  
 ~GRAPH( );  
 int V ( ) const;  
 int E( ) const;  
 int insert (Edge \*);  
 int remove (Edge \*);  
 Edge \* edge(int, int);  
};

//АЛГОРИТМ отыскания MST(в конструкторе)template <class Graph, class Edge>  
class MST {  
 const Graph &G;  
 vector <double> wt; // веса  
 vector <Edge \*> fr, mst; //кандидаты и результат  
public:  
 MST(const Graph &\_G) : G(\_G), mst(G.V( )), wt(G.V( )), fr(G.V( ))  
 { int min = -1;  
 for (int v = 0; min != 0; v = min)  
 { min = 0;  
 for (int w = 1; w < G.V(); ++w)  
 if (mst[w] == 0)  
 { double P; Edge \* e = G.edge(v, w);  
 if (e) //ребро существует  
 if ((P = e->wt()) < wt[w]) {wt[w] = P; fr[w] = e;}  
 if(wt[w] < wt[min]) min = w;  
 }  
 if(min) mst[min] = fr[min];  
 }  
 }   
 void show( ) //Вывод результата  
 { for (int v = 1; v < G.V( ); ++v) if(mst[v]) mst[v]->show( ); }  
};

*Пример* работы алгоритма Прима. Используется тот же граф, что и в предыдущих примерах (рис.30.12, 31.1). В качестве стартовой взята вершина «*f*».



*Рис*.*31.8.* Первый шаг. Кайма = <{a,b},{a,c},{c,f},{f,g}>.



*Рис*.*31.9*. Второй шаг. Кайма = <{a,c},{b,c},{c,f},{f,g},{b,e}>.



*Рис*.*31.10*. Третий шаг. Кайма = <{c,g},~~{b,c}~~,~~{c,f}~~,{f,g},{b,e},{c,d}>.



*Рис*.*31.11.* Четвёртый шаг. Кайма = <{d,g},{g,h},~~{f,g}~~,{b,e},{c,d}>.



*Рис*.*31.12*. Пятый шаг. Кайма = <{d,e},{g,h},{b,e},~~{c,d}~~>.



*Рис*.*31.13*. Шестой шаг. Кайма = <{e,h},{g,h},~~{b,e}~~>.



*Рис*.*31.14*. Седьмой шаг (результат).

### 31.3. Матроиды трансверсалей: трансверсаль с наибольшим весом

Дано множество *E* и семейство его подмножеств *Q* = (*A*1, … *An*), не обязательно непересекающихся, не обязательно различных, и частичная трансверсаль *S* ⊆ *E*., т. е. существует инъективное отображение *f* : *S* → {1, … *n*}, что *e*∈ *Af*(*e*) для каждого *e* ∈ *S*.

Если ℑ — семейство частичных трансверсалей *S* семейства *Q*, то *M*(*Q*) = <*E,*  ℑ > является матроидом (*матроидом трансверсалей*).

Жадный алгоритм для матроида трансверсалей отыскивает частичную трансверсаль с наибольшим весом.

Предполагаем, что элементам множества присвоены некоторые веса, и это множество упорядочено по не возрастанию весов (неотрицательных):

*E* = { *e*1, … *em*}.

*Данные*: Семейство *Q* = (*A*1, … *An*) подмножеств множества *E* = {*e*1, … *em*}, представленное с помощью двудольного графа *H*. Элементы *E* упорядочены.

*Результат*: частичная трансверсаль *S* с наибольшим весом (и паросочетание *M* графа *H,* определяющее функцию *f* : *S* → {1, … *n*}).

*S* = ∅; *M* = ∅;  
for (int *i* = 1; *i* ≤ *m*; ++*i*)  
 if (существует в *H* чередующаяся цепь *P* относительно *M* с началом в *ei*)  
 { *S* = (*S* ⋃ { *ei* }); *M*  = *M* ⨂ *P* ; }

До начала работы алгоритма все вершины *ei* свободны. От каждой из них в порядке очереди по убыванию веса строится чередующаяся цепь, увеличивающая паросочетание на одно ребро и мощность частичной трансверса­ли на 1. Процесс заканчивается по получении совершенного паросочетания (полной трансверсали) или отсутствии возможности построить очередную чередующуюся цепь.

*Пример* работы алгоритма получения трансверсали с наибольшим весом.



*Рис*.*31.15*. Исходный двудольный граф.



*Рис*.*31.16*. Результат первых трёх шагов алгоритма.



*Рис*.*31.17*. Чередующаяся цепь на четвёртом шаге.



*Рис*.*31.18*. Пятый шаг.



*Рис*.*31.19*. Шестой шаг. Получена полная трансверсаль.

Если известно, что должна получиться полная трансверсаль, то жадный алгоритм ни к чему, потому что алгоритм Хопкрофта-Карпа получит то же самое решение, использовав для этого не *n*, а *n1/2* итераций. Смысл жадного алгоритма — оптимальное решение после каждой итерации для обработанной части множества *E*.

Контрольные вопросы

1. Задано множество мощностью 50 в виде последовательности, упорядоченной по убыванию веса, и 100 его подмножеств суммарной мощностью 2500. Сколько итераций потребуется алгоритму для отыскания частичной трансверсали с наибольшим весом?
2. Дана система из m линейных алгебраических уравнений с n неизвестными (n < m), упорядоченная по убыванию веса. Какова временная сложность отбора подсистемы из n независимых уравнений с наибольшим весом?
3. Задан граф из n вершин и m рёбер в виде матрицы инциденций со столбцами-рёбрами, упорядоченными по неубыванию веса. Какова временная сложность отыскания в этом графе стягивающего дерева наименьшей стоимости?
4. Задан граф из 100 вершин и 1000 рёбер. Рёбра нагружены весом, не превышающим 10. Сколько рёбер будет содержать стягивающее дерево наименьшей стоимости?
5. Какой алгоритм рекомендуется для разреженного графа с нагруженными рёбрами для отыскания в нём дерева наименьшей стоимости?
6. Какой алгоритм рекомендуется для насыщенного графа с нагруженными рёбрами для отыскания в нём дерева наименьшей стоимости?

# 32. Жадные алгоритмы для заведомо переборных задач

## 32.1. Общая схема жадного и переборного алгоритмов

**Жадный алгоритм**

1. Упорядочить входные множества по заданному критерию — создать очередь.
2. Отбирать из полученной очереди элементы и добавлять их в решение, пропуская те, которые являются линейной комбинацией уже отобранных.

В чистом виде эта схема реализована в алгоритме Краскала: отобранный элемент становится частью решения и больше не обрабатывается. Попытка обобщить эту схему привела к созданию теории матроидов.

Другие алгоритмы отличаются отдельными деталями. Так, матричный алгоритм на каждом шаге пересчитывает входные вектора. Появление 0 на главной диагонали означает, что очередной вектор зависим, и его нужно пропустить. Алгоритм для трансверсалей на каждом шаге может перестроить полученное ранее паросочетание.

Алгоритм Прима создаёт очередь из ребёр, инцидентных стартовой вершине и пересматривает её на каждом шаге, убирая элементы, ставшие зависимыми, и добавляя новые. Алгоритм Дейкстры пересчитывает оценки расстояний для ещё не обработанных вершин.

**Переборный алгоритм**

Для заведомо переборных задач применяется **недетерминированый** алгоритм:

1. Угадать (сгенерировать) решение.
2. Проверить.

Шаги повторяются, пока не будет получено решение или исчерпано время.

Для угадывания решения можно применить генератор случайного подмножества или случайной перестановки.

Вместо угадывания можно просто подать на вход алгоритма все возможные наборы данных и проверить каждый — применить **полный перебор**. Для этого можно воспользоваться генератором всех подмножеств универсума, всех подмножеств заданной мощности, всех перестановок. Все генераторы были рассмотрены в теме «Генерация тестов».

Переборный алгоритм универсален. Если задача имеет много решений, они будут найдены все. Если ни одного решения не найдено, значит его не существует. Поэтому переборный алгоритм может служить критерием качества и работоспособности алгоритмов полиномиальной сложности.

Но трудоёмкость алгоритма такова, что его можно применить только для задач небольшой размерности. Все усовершенствования имеют цель увеличить доступную для решения размерность задачи.

Одно из таких усовершенствований — **алгоритм перебора с возвратом**: решение строится в виде расширяемого вектора, для каждого элемента которого задано конечное множество допустимых значений и функция для отбраковки заведомо негодных. Для некоторых задач этот подход известен как «Метод ветвей и границ».

Другое усовершенствование — **жадный подход**: проверяемые входы предварительно упорядочиваются по какому-то критерию, например, по убыванию перспективности. Детали может подсказать конкретная задача. Смысл подхода: увеличить вероятность того, что решение будет найдено достаточно быстро и не потребует перебора всех возможных альтернатив.

**Эмпирический алгоритм**: получить пусть не самый лучший, но достаточно хороший результат за разумное время, не применяя полного перебора.

## 32.2. Раскладка по ящикам

Задача: упаковать набор предметов в минимальное число ящиков фиксированного размера.

Алгоритм:

void FirstFind (int size[ ], int N, int SIZE, int bin[ ])  
// *size*[ ] — набор предметов (размеры), *N* — их количество;  
// *SIZE* — размер ящика, *bin*[ ] — результат раскладки  
{ int Used[NN] = {0}; //Результат  
 for (int item = 0; item < N; ++item)  
 { int binLoc = 0;  
 while (Used[binLoc] + size[item] > SIZE) ++binLoc;  
 bin[item] = binLoc;  
 Used[binLoc] += size[item];  
 }  
}

Отличие от жадного алгоритма для матроидов: результат неудачного выбора ящика, возможно, будет препятствием для оптимального распределения предметов и нет разумного способа избежать этого.

*Пример*: Ящик 10, предметы 5, 7, 3, 9, 6, 8, 1, 4, 2, 5.

*Первый подход*: обработать исходную (случайную) последовательность предметов: упаковывать предмет в первый подходящий по порядку ящик.

*Результат*: 5-3-1, 7-2, 9, 6-4, 8, 5 (7 ящиков).

*Оптимум*: 9-1, 8-2, 7-3, 6-4, 5-5.

*Второй подход*: (жадный алгоритм) то же, сперва упорядочить вход по неубыванию.

*Предметы*: 1, 2, 3, 4, 5, 5, 6, 7, 8, 9.

*Результат*: 1-2-3-4, 5-5, 6, 7, 8, 9 (6 ящиков).

*Третий подход*: (жадный алгоритм) то же, сперва упорядочить вход по невозрастанию.

*Предметы*: 9, 8, 7, 6, 5, 5, 4, 3, 2, 1.

*Результат* = *оптимум*: 9-1, 8-2, 7-3, 6-4, 5-5 (5 ящиков).

В общем случае: неупорядоченная последовательность — перерасход ящиков в 70%, упорядоченная — 50%.

## 32.3. Упаковка рюкзака

Задача: дан набор предметов разных размеров и разной стоимости. Упаковать в рюкзак фиксированного размера набор предметов максимальной стоимости.

*Пример*: Рюкзак 80, набор (размер-стоимость) 25-50, 20-80, 20-50, 15-45, 30-105, 35-35, 20-10, 10-45.

*Стратегия*:

— рассчитаем удельные стоимости: 2, 4, 2.5, 3, 3.5, 1, 0.5, 4.5;

— упорядочим: 10-45, 20-80, 30-105, 15-45, 20-50, 25-50, 35-35, 20-10.

— просматриваем упорядоченную последовательность и заполняем рюкзак, отбрасывая то, что не входит:

*Результат*: 10+20+30+15=75 (стоимость 275).

*Оптимум*: 10+20+30+20=80 (стоимость 280).

## 32.4. Упаковка рюкзака: точное решение

Пусть для простоты стоимость равна объёму (задача на максимальную сумму элементов подмножества, укладывающуюся в заданный предел). Особенность задачи: оптимум известен! Мы знаем, когда можно остановиться. Идея алгоритма: комбинация жадного подхода с полным перебором в расчёте наткнуться на результат много раньше исчерпания всех возможных решений.

*Дана* последовательность: 27, 22, 14, 11, 7, 1. Рюкзак = 55.

1. Применяем жадный алгоритм: 27+22+1 = 50.
2. Отбираем по одному, к остальным — жадный алгоритм:

27 +22+1 = 50  
14 +27+11+1 = 53  
11 + 27+14+1 = 53  
7 +27+14+1 =49  
1 +27+22 = 50

1. Отбираем все комбинации по 2, дополняем:

27+22 +1 = 50  
27+14 +11+1 = 53  
27+11 +14+1 = 53  
27+7 +14+1 =49  
27+1 +22 = 50  
22+14 +11+7+1 =55 — **оптимум найден**!  
22+11 +14+7+1 = 55  
22+7 +14+11+1 = 55  
22+1 +27 = 50  
14+11 +27+1 = 53  
14+7 +27+1 = 49  
14+1 +27+11 = 53  
11+7 +27+1 = 46  
11+1 +27+14 = 53  
7+1 +27+14 = 49.

Очевидно, что алгоритм переберёт все комбинации. Можно остановиться, когда результат устроит или будет исчерпано разумное время.

## 32.5. Алгоритмы полиномиальной сложности и их контроль перебором

Задача: отыскание максимальной клики.

**Эмпирический алгоритм полиномиальной сложности**

*Данные*: *a*[*n*][*n*] — матрица смежности графа из *n* вершин.

Результат: *K*[*maxv*] — клика мощностью *maxv*.

for (auto i = 0; i < N; ++i)

{ K[ 0 ] = i;

for(auto st = i + 1; st < N; ++st)

if(a[ i ][ st ])

{ k = 1;

for(auto j = st; j < N; ++j)

{ num = 1;

while((a[ K[ num-1 ] ][ j ]) && (num <= k)) ++num;

if ((num - 1) == k)

{ ++k; K[ k-1 ] = j; }

}

if (k > maxv) //Зафиксировать решение

{ maxv = k;

for(auto i1 = 0; i1 < k; ++i1) ans[ i1 ] = K[ i1 ] + 1;

}

if (k == maxv) { //... и выдать

cout << "\n max=" << maxv << " : ";

for (auto i1 = 0; i1 < maxv; ++i1)

cout << (K[i1] + 1) << " ";

cin.get();

} } }

cout << "\n Клика мощностью " << maxv <<" из вершин: ";

for(auto i = 0; i < maxv; ++i)

cout << ans[ i ] << " ";

**Альтернативный переборный алгоритм**

void G(int k) //Перебор с возвратом

{ int i, i0;

if(k == 1) i0 = 0; else i0 = K[ k-2 ] + 1;

for( i = i0; i < N; i++)

if (U[ i ]) {

K[ k-1 ] = i; j = 0;

while ((j < k) && a[ K[ j ] ][ i ]) ++j;

if (j+1 == k) { //Найдена клика...

if (k > maxv) { //больше предыдущей, зафиксировать решение

maxv = k;

for (auto i1 = 0; i1 < k; ++i1)

ans[ i1 ] = K[ i1 ] + 1;

}

if (k == maxv) { //... и выдать

cout << '\n' << " max=" << maxv << " : ";

for(auto i1 = 0; i1 < maxv; ++i1)

cout << (K[ i1 ] + 1) << " ";

cin.get( );

}

U[ i ] = 0; //Вершина ( *i* ) теперь занята

G(k+1); // Попробовать расширить решение

U[ i ] = 1; //Возврат: ( *i* ) снова свободна

}

}

}

*Результат прогона*: при малых значениях *n* результаты совпадают. Расхождения начинаются при *n* > 10. Подробнее — см. «Пользовательские структуры данных», с. 47–49.

## Приближённая раскраска графа

Число красок, получаемое наилучшим полиномиальным алгоритмом, в общем случае более чем вдвое превышает оптимальное. Алгоритмы полиномиальной сложности существуют для частных случаев, например, для планарных графов. В этом случае хроматическое число равно 4. Для пустых графов хроматическое число равно 1, для деревьев — 2, для полных графов — количеству вершин *n*.

Интерес представляют насыщенные неполные графы.

*Пример* алгоритма полиномиальной сложности.

1. Упорядочить вершины по возрастанию степени.
2. Раскрасить:

for (auto v : V) color[v] = 0;  
for (auto v : V) {  
 int c = 1; bool f = false;  
 for (auto u : List[v]) f ||= (color[v] == c);  
 if( f ) ++c;  
 color[v] = c;  
 }

*Результат*: раскраска числом цветов, равным максимальной степени вершины + 1.

## Задача о коммивояжёре

Модификация алгоритма Дейкстры для графов с неотрицательными весами: ищем кратчайший путь через все вершины:

— перебираем рёбра в порядке возрастания весов;

— добавляем в путь ребро, если выполнены условия: ребро не замыкает цикл (не полный) или не является ответвлением.

*Пример:* Граф с весовой матрицей

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **От/до** | **1** | **2** | **3** | **4** | **5** | **6** |
| **0** | 16 | 12 | 13 | 7 | 6 | 11 |
| **1** |  | 21 | 18 | 8 | 19 | 5 |
| **2** |  |  | 20 | 1 | 3 | 15 |
| **3** |  |  |  | 14 | 10 | 4 |
| **4** |  |  |  |  | 2 | 17 |
| **5** |  |  |  |  |  | 9 |

Для упрощения анализа веса рёбер заменены их порядковыми номерами в упорядоченной последовательности.



*Рис*. *32.1.* Граф к задаче коммивояжёра

*Жадный алгоритм*: отбор рёбер из последовательности, упорядоченной по возрастанию веса, с пропуском рёбер, замыкающих цикл или ответвлений.

2-4(1); 4-5(2) → берём;  
2-5(3) — образует цикл;  
3-6(4); 1-6(5); 0-5(6) → берём;  
0-4(7) — образует цикл;  
1-4(8) — ответвление;  
5-6(9) — ответвление;  
3-5(10) — ответвление;  
0-6(11) — ответвление;  
0-2(12) — образует цикл;  
0-3(13) → берём;  
3-4(14) — образует цикл;  
2-6(15) — ответвление;  
0-1(16) — ответвление;  
4-6(17) — ответвление;  
1-3(18) — ответвление;  
1-5(19) — ответвление;  
2-3(20) — ответвление;  
1-2(21) → берём.

*Результат*: 0-3-6-1-2-4-5-0, длина 13+4+5+21+1+2+6 = 52 (рис. 15.2).



*Рис*. 32.2. Результат работы жадного алгоритма

Есть вариант 0-3-6-1-4-2-5-0 с длиной 13+4+5+8+1+3+6 = 40 (рис. 32.3).



*Рис*. *32.3*. Оптимальный результат

## 32.6. Муравьиный алгоритм для задачи коммивояжёра

Задача формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с *n* вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжёр, а веса рёбер отражают расстояния (длины) или стоимости проезда. Эта задача является *NP*-трудной, и точный переборный алгоритм её решения имеет факториальную сложность.

Моделирование поведения муравьёв связано с распределением феромона на тропе – ребре графа в задаче коммивояжёра. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его рёбрах, следовательно, большее количество муравьёв будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости – большинство муравьёв двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с другой стороны, большое время испарения может привести к получению устойчивого локального оптимального решения.

Теперь с учётом особенностей задачи коммивояжёра, мы можем описать локальные правила поведения муравьёв при выборе пути.

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещённых городов – список запретов. Обозначим через *Ji,k* список городов, которые необходимо посетить муравью *k*, находящемуся в городе *i*.

2. Муравьи обладают «зрением» – видимость есть эвристическое желание посетить город *j*, если муравей находится в городе *i*. Будем считать, что видимость обратно пропорциональна расстоянию между городами *ηi,j* = 1/*Di,j*.

3. Муравьи обладают «обонянием» – они могут улавливать след феромона, подтверждающий желание посетить город *j* из города *i* на основании опыта других муравьёв. Количество феромона на ребре <*i*, *j*> в момент времени *t* обозначим через *τ*i,j(*t*).

4. На этом основании мы можем сформулировать вероятностно-пропорцио­наль­ное правило, определяющее вероятность перехода *k*-ого муравья из города *i* в город *j*:

*Pijk*(*t*) = [τij(*t*)]α . [*η* ij]β / [τij(*t*)]α . [*η* ij]β, *j* ∈ *Jik*;

*Pijk*(*t*) = 0, *j* ∉ *Jik*  (1)

где *α*, *β* – параметры, задающие веса следа феромона. При *α* = 0 алгоритм вырождается до жадного (будет выбран ближайший город). Заметим, что выбор города является вероятностным, правило (1) лишь определяет ширину зоны города *j*; в общую зону всех городов бросается случайное число, которое и определяет выбор муравья. Правило (1) не изменяется в ходе алгоритма, но у двух разных муравьёв значение вероятности перехода будут отличаться, т. к. они имеют разный список разрешённых городов *Jik*.

5. Пройдя ребро <*i, j*>, муравей откладывает на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть *Tk* ( *t* ) есть маршрут, пройденный муравьём *k* к моменту времени *t*, *Lk* ( *t* ) – длина этого маршрута, а *Q* – параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде

Δ *τ ijk* ( *t* ) = *Q / Lk* ( *t* ) для <*i*, *j*> ∈ *Tk* ( *t* ), иначе 0.

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть *p* ∈ [0, 1] есть коэффициент испарения, тогда итоговое количество феромона на ребре <*i*, *j*> с учётом испарения имеет вид

*τij*( *t* + 1 ) = (1 – *p*) *τij*( *t* ) + Δ *τ ij* ( *t* ), (2)

где Δ *τ ij* ( *t* ) – общее количество феромона, отложенное на ребре <*i*, *j*> всеми муравьями в колонии в момент *t*.

В начале алгоритма количества феромона на рёбрах принимается равным небольшому положительному числу. Общее количество муравьёв остаётся постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

Дополнительная модификация алгоритма (ускорение его работы) может состоять в ведении так называемых «элитных» муравьёв, которые усиливают рёбра наилучшего маршрута, найденного с начала работы алгоритма. Обозначим через *T\** наилучший текущий маршрут, через *L\** – его длину. Тогда если в колонии есть *e* элитных муравьёв, то рёбра маршрута получат дополнительное количество феромона

Δ*τe* = *e* . *Q* / *L\**  (3)

Алгоритм в целом выглядит следующим образом.

1. Получение матрицы расстояний *D*, расчёт матрицы *η* и выбор *α*, *β*, *e* и *Q*.
2. Размещение *m* муравьёв по городам случайным образом без совпадений.
3. Инициализация рёбер начальной концентрацией феромона.
4. Выбор начального маршрута *T\** и определение его длины *L\**.
5. Цикл по времени жизни колонии:

* цикл по муравьям: рассчитать для каждого вероятность выбора очередного города, сделать случайный выбор, получить маршрут *Tk* и определить его длину *Lk*;
* если найдено лучшее решение, обновить *T\** и *L\**;
* обновить значение феромона на рёбрах по правилам (2) и (3).

Время работы алгоритма пропорционально времени жизни колонии, количеству городов *n* и муравьёв *m*.

*Контрольные вопросы*

1. Имеется набор грузов различного объёма и стоимости. В каком порядке следует укладывать грузы в рюкзак заданного объёма, чтобы стоимость поместившихся грузов была максимальна?
2. Имеется набор посылок, отличающихся размером: <5, 7, 3, 9, 6, 8, 1, 4, 2, 5, 3, 6, 8, 2, 1, 5, 7, 2>. В каком порядке следует их упаковывать в ящики размером 10, чтобы использовать минимальное количество ящиков?

**СОДЕРЖАНИЕ**

1. Цели и задачи курса 3

2. Оценка временной сложности алгоритмов 12

3. Понятие данных. Примитивы и структуры. Стандартные структуры данных 21

4. Множества в памяти ЭВМ 33

4.1. Представление множества набором элементов 33

4.2. Представление множества отображением на универсум 41

4.3. Замечание о функциях — операциях над множествами 49

5. Генерация тестов 54

*5.1. Генерация случайного подмножества* 55

*5.2. Генерация последовательности всех подмножеств заданного множества* 58

*5.3. Множества с повторениями. Тест — аналог кода Грея* 60

*5.4. Случайное подмножество заданной мощности* 63

*5.5. Последовательность всех подмножеств заданной мощности* 66

*5.6. Генерация перестановок* 71

6. Множество — пользовательский тип данных 73

7. Типы классов и служебные функции-члены 81

8. Перегрузка операций 120

9. Перехват управления памятью: перегрузка new и delete 145

10. Понятие о шаблонах. Стандартная библиотека шаблонов 166

11. КОНТЕЙНЕРНЫЕ КЛАССЫ 173

12. Вспомогательные средства STL 187

13. Интеллектуальные указатели 201

Ещё об интеллектуальных указателях 213

14. Бинарное отношение на множестве. Графы 214

15. ДЕРЕВЬЯ 221

16. Нерекурсивные алгоритмы обхода дерева 241

17. Альтернативные способы хранения дерева в памяти: вектор пар и вектор битов 244

18. Обходы графов 254

19. Стягивающие деревья 262

Cвойства глубинного и ширинного СД 265

20. Фундаментальные циклы 266

Алгоритм нахождения множества фундаментальных циклов 269

21. Двусвязность и сильная связность 281

22. Сильная связность в орграфе 298

23. Эйлеров цикл 312

24. Гамильтонов цикл 319

25. Переборные алгоритмы на графах 328

26. Алгоритмы на графах с нагруженными вершинами. Сортировка 333

26.1. Дерево сортировки и сортировка деревом 336

26.2. Сортировка слиянием 349

26.3. Быстрая сортировка 352

26.4. Цифровая сортировка 354

26.5. Cортировка цепочек 359

27. Алгоритмы на графах с нагруженными рёбрами (Кратчайшие пути и смежные вопросы) 368

27.1. Кратчайший путь от фиксированной вершины. Общий случай 374

27.2. Алгоритм Дейкстры 379

27.3. Кратчайший путь в бесконтурном графе 383

27.4. Алгоритм Флойда 385

27.5. Транзитивное замыкание отношения (алгоритм Уоршалла) 391

28. Алгоритмы на графах с нагруженными рёбрами и вершинами. Потоки в сетях 393

29. Паросочетания в двудольных графах 428

29.1. Основные понятия 428

29.2. Алгоритм Хопкрофта – Карпа для отыскания наибольшего паросочетания в двудольном графе 435

29.3. Теория трансверсалей 446

29.4. Вершинное покрытие в двудольном графе 449

30. Жадные алгоритмы решения оптимизационных задач 455

30.1. Понятие о матроиде 455

30.2. Жадный алгоритм для матроидов (теорема Рамо — Эдмонса) 460

30.3. Матричные матроиды: независимое множество векторов 461

30.4. Графовые матроиды: стягивающее дерево наименьшей стоимости 471

31. Эффективные алгоритмы для отыскания стягивающего дерева наименьшей стоимости и матроиды трансверсалей 483

31.1. Алгоритм Краскала 483

31.2. Алгоритм Прима 488

31.3. Матроиды трансверсалей: трансверсаль с наибольшим весом 497

32. Жадные алгоритмы для заведомо переборных задач 505

32.1. Общая схема жадного и переборного алгоритмов 505

32.2. Раскладка по ящикам 509

32.3. Упаковка рюкзака 511

32.4. Упаковка рюкзака: точное решение 512

32.5. Алгоритмы полиномиальной сложности и их контроль перебором 514

Приближённая раскраска графа 518

Задача о коммивояжёре 519

32.6. Муравьиный алгоритм для задачи коммивояжёра 525

Колинько Павел Георгиевич

**Пользовательские структуры данных**

Конспект лекций

Издание публикуется в авторской редакции

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Подписано в печать . .2020 . Формат 60×84 1/16.

Бумага офсетная. Печать цифровая. Печ. л. 22,0.

Гарнитура «Times New Roman». Тираж экз. Заказ

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Издательство СПбГЭТУ «ЛЭТИ»

197376, С.-Петербург, ул. Проф. Попова, 5