СРЕДНЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

Т. Г. Чурина, Т. В. Нестеренко

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Учебное пособие для СПО

Рекомендовано Учебно-методическим отделом СПО в качестве учебного пособия для использования в учебном процессе образовательными учреждениями среднего профессионального образования по укрупненным группам специальностей 09.00.00 «Информатика и вычислительная техника», 10.00.00 «Информационная безопасность»

Москва
Ай Пи Ар Медиа
2020

Саратов
Профобразование
2020

УДК 519.85
ББК 32.97
Ч-93

Рецензент:
Бульонков М. А. — канд. физ.-мат. наук

Чурина, Т. Г.
Основы алгоритмизации и программирования : учебное пособие для СПО / Т. Г. Чурина, Т. В. Нестеренко ; Новосибирский государственный университет. — Эл. изд. — Москва : Ай Пи Ар Медиа ; Саратов : Профобразование, 2020. — 214 с. — (Среднее профессиональное образование)

ISBN 978-5-4497-0465-8 (Ай Пи Ар Медиа)
ISBN 978-5-4488-0802-9 (Профобразование)

В учебном пособии представлены абстрактные типы данных, описание алгоритмов обхода графов и деревьев в глубину и в ширину, нахождения путей в графе, различных видов циклов, определение максимального потока в графе и поиск паросочетаний. Приведены алгоритмы построения каркасов графа, алгоритмы с возвратом, рассмотрены задачи, решения которых опираются на использование этих алгоритмов.

Учебное пособие предназначено для изучения дисциплины «Основы алгоритмизации и программирования» по укрупненным группам специальностей среднего профессионального образования 09.00.00 «Информатика и вычислительная техника», 10.00.00 «Информационная безопасность».

Учебное электронное издание

Для создания электронного издания использовано:
Приложение pdf2swf из ПО Swftools, ПО IPRbooks Reader, разработанное на основе Adobe Air

УДК 519.85
ББК 32.97

© Чурина Т. Г., Нестеренко Т. В., 2014
© Новосибирский государственный университет, 2014
© Оформление. ООО Компания «Ай Пи Ар Медиа», «Профобразование», 2020

ОГЛАВЛЕНИЕ

Введение 7
Глава 1. Динамическая память 8
1.1 Средства работы с динамической памятью 8
1.1.1 Функции управления памятью 9
1.1.2 Указатели 10
1.1.3 Статические и динамические переменные 15
1.2 Списки 16
1.2.1 Односвязные списки 16
1.2.2 Двусвязные списки 20
1.2.3 Циклические списки 22
1.2.4 Иерархические списки 23
1.3 Простые структуры данных 23
1.3.1 Стеки 24
1.3.2 Задача перевода выражения из инфиксной формы в постфиксную 27
1.3.3 Очереди 31
Глава 2. Графы 34
2.1 Отношения 34
2.2 Основные определения 36
2.3 Представление графов в памяти машины 39
2.3.1 Матрица смежности 39
2.3.2 Матрица инцидентности 40
2.3.3 Списки смежности 40
2.3.4 Табличное представление списков смежностей 41
2.4 Топологическая сортировка графа 42
2.4.1 Реализация алгоритма на матрице смежности 43
2.4.2 Реализация алгоритма на иерархических списках 44
2.5 Обход графа в глубину 47
2.5.1 Свойства поиска в глубину 50
2.5.2 Поиск в глубину в ориентированном графе 51
2.5.3 Решение задачи топологической сортировки методом поиска в глубину 52
2.6 Обход графа в ширину 54
2.6.1 Метод поиска в ширину 54
2.6.2 Нахождение кратчайшего пути в лабиринте 58
2.7 Раскраска графов 60
2.7.1 Задача составления расписаний 61
2.7.2 Задача распределения ресурсов 62
2.7.3 Задача экономии памяти 62
2.7.4 Алгоритм последовательной раскраски 63
2.7.5 Проблема четырех красок 65

2.7.6 Раскраска ребер 65
Глава 3. Деревья 67
3.1 Бинарные деревья 68
3.2 Обходы деревьев 69
3.2.1 Обходы деревьев в глубину 69
3.2.2 Обход деревьев в ширину 71
3.3 Представление деревьев 73
3.3.1 Скобочные представления деревьев 73
3.3.2 Представление дерева списком прямых предков 74
3.3.3 Реализация бинарных деревьев на Си 74
Глава 4. Переборные задачи и динамическое программирование 76
4.1 Алгоритмы с возвратом 76
4.1.1 Задача о ходе коня 77
4.1.2 Задача о кубике 80
4.1.3 Задача о стабильных браках 81
4.1.4 Метод ветвей и границ 84
4.1.5 Использование метода ветвей и границ для решения задачи о рюкзаке 85
4.2 Динамическое программирование 86
4.2.1 Задача о наибольшей общей подпоследовательности 88
4.2.2 Задача о преобразовании строк 92
4.2.3 Задача о рюкзаке 96
4.2.4 Задача о горнолыжных соревнованиях 99
Глава 5. Структуры данных поиска 102
5.1 Деревья двоичного поиска 102
5.2 Сбалансированные деревья 103
5.3 B-деревья 106
5.3.1 Алгоритм поиска 108
5.3.2 Разбиение вершины B-дерева 110
5.3.3 Добавление элемента в B-дерево 111
5.3.4 Удаление элемента из B-дерева 113
5.4 Дерево отрезков 115
5.4.1 Создание дерева отрезков 115
5.4.2 Запрос суммы на отрезке 116
5.4.3 Запрос обновления 119
5.5 Дерево Фенвика 119
5.6 Кучи (пирамиды) 123
5.6.1 Бинарные кучи 123
5.6.2 *K*-ичные кучи 126
5.6.3 Фибоначчиевы кучи 127
5.7 Очередь с приоритетами 128
5.8 Декартовы деревья 130

5.8.1 Построение декартова дерева 139
5.8.2 Использование декартова дерева... 140
5.9 Система непересекающихся множеств (СНМ) 144
5.9.1 Простая реализация 145
5.9.2 Реализация с помощью списков 145
5.9.3 Весовая эвристика.. 146
5.9.4 Реализация с использованием дерева.................................... 147
5.9.5 Эвристика объединением по рангу 148
5.9.6 Эвристика сжатия путей... 148
Глава 6. Пути в графе.. 151
6.1 Нахождение кратчайшего пути из одного источника 151
6.2 Алгоритм Беллмана–Форда 154
6.3 Нахождение кратчайших путей между всеми парами вершин 155
6.4 Транзитивное замыкание графа 157
6.5 Связность в графе 158
6.5.1 Компоненты связности... 158
6.5.2 Двусвязность 160
6.5.3 Нахождение двусвязных компонент и точек сочленения .. 162
6.6 Построение минимального каркаса графа................................. 165
6.6.1 Алгоритм Краскала 165
6.6.2 Алгоритм Прима 169
6.6.3 Алгоритм Прима с удалением ребер 172
6.7 Циклы в графе.. 175
6.7.1 Эйлеровы циклы.. 175
6.7.2 Алгоритмы поиска эйлерова цикла 178
6.7.3 Гамильтоновы циклы.. 182
Глава 7. Потоки в сетях 185
7.1 Сокращение потоков между вершинами................................... 187
7.2 Несколько истоков... 188
7.3 Остаточная сеть 189
7.4 Дополняющие пути 191
7.5 Разрезы в сетях 192
7.6 Метод Форда-Фалкерсона 194
7.6.1 Пример 195
7.7 Задача о максимальном паросочетании в двудольном графе .. 198
7.8 Алгоритм проталкивания предпотока 200
7.8.1 Высотная функция 201
7.8.2 Операция проталкивания предпотока.................................... 201
7.8.3 Операция поднятия вершины 202
7.8.4 Описание алгоритма 202
7.8.5 Корректность метода 204
7.8.6 Пример 205

Заключение 213
Список литературы... 214

ВВЕДЕНИЕ

В учебных пособиях [9, 10] рассмотрены такие темы, как традиционные методы программирования, вопросы, касающиеся основ арифметики в позиционных системах счисления, представления и кодирования числовой информации, а также такие важные понятия, как перестановки, поиск и сортировка.

В данном учебном пособии особое внимание уделяется представлению абстрактных типов данных, среди которых рассмотрены стеки, очереди, деревья, графы, отмечены элементы их реализации на компьютере. В пособие включено описание алгоритмов обхода графов и деревьев в глубину и в ширину, нахождения путей в графе, различных видов циклов, определение максимального потока в графе и поиск паросочетаний. Также рассмотрены решения задач, использующих знание таких структур данных, как B-дерево, дерево отрезков и дерево Фенвика. Обозначены проблемы, касающиеся связности графа. Приведены алгоритмы построения каркасов графа, алгоритмы с возвратом, рассмотрены задачи, решения которых опираются на использование этих алгоритмов. В пособии приведены примеры задач и способы их решения с использованием метода динамического программирования. По основным алгоритмам приведена оценка сложности, при этом используются O-обозначения для выражения меры эффективности алгоритма, в первую очередь для времени его исполнения.

Для описания алгоритмов используются следующие обозначения:

$x \leftarrow y$; – значение выражения y записать в переменную x;

| – начало комментариев в описании алгоритмов, комментарии заканчиваются концом строки.

Определения в тексте пособия выделены двойной вертикальной чертой.

Целью данного пособия является знакомство аудитории с широким спектром различных алгоритмов, поэтому доказательства корректности многих из них оставлены в виде ссылок на источники.

Авторы выражают благодарность В. В. Мухортову и М. А. Бульонкову за конструктивные замечания и полезные предложения, которые повысили качество предложенного материала.

ГЛАВА 1. ДИНАМИЧЕСКАЯ ПАМЯТЬ

Память для хранения данных может выделяться как статически, так и динамически. В первом случае память выделяется в процессе компиляции программы. В соответствии с типом встретившегося объекта компилятор вычисляет объем памяти, требуемый для его размещения, и определяет место, где эти объекты будут располагаться. Это может быть сегмент данных либо стек.

Часто возникают ситуации, когда заранее не известно, сколько объектов и каких размеров будет хранить программа. В этом случае используется динамическое выделение памяти. Оно выполняется в процессе исполнения программы. Управление динамической памятью – это способность определять размер объекта и выделять для его хранения соответствующую область памяти в процессе исполнения программы.

1.1 Средства работы с динамической памятью

При динамическом выделении памяти для хранения данных используется специальная область памяти, так называемая *куча (heap)*. Объем кучи и ее местоположение зависят от модели памяти, которая определяет логическую структуру памяти программы. При каждом обращении к функции распределения памяти выделяется запрошенное число байт. Адрес начала выделенной памяти возвращается в точку вызова функции и записывается в переменную-указатель. Указатель – это переменная, значением которой является адрес области памяти (рис. 1.1).

Рис. 1.1. Связь указателя с динамической переменной

Созданная таким образом переменная называется *динамической переменной*. Дальнейшая работа с выделенной областью осуществляется через переменную-указатель, хранящую адрес выделенной области памяти.

В языке программирования Си объявлен макрос **`NULL`**, значением которого является зависящая от реализации константа нулевого указателя. *Константа нулевого указателя* – это целочисленное константное выражение со значением 0, или такое же выражение, но приведенное к типу **`void *`**. Константа нулевого указателя, приведенная к любому типу указателей,

является *нулевым указателем*. Нулевой указатель не равен указателю на любой объект или функцию.

Нулевые указатели придуманы как удобный способ «отметить» указатели, которые заведомо не указывают на корректный адрес в памяти. Например, при объявлении указателя как автоматической переменной его значение не определено. Чтобы отметить, что этот указатель еще не содержит корректный адрес в памяти, такому указателю присваивают константу нулевого указателя.

1.1.1 Функции управления памятью

Мы будем придерживаться **ANSI C** – стандарта языка Си, опубликованного Американским национальным институтом стандартов (ANSI). Следование этому стандарту помогает создавать легко переносимые программы.

Распределение памяти в Си следующее. Для глобальных переменных отводится фиксированное место в памяти на все время работы программы. Локальные переменные хранятся на стеке. Функции **malloc()**, **calloc()**, **realloc()** и **free()** используются для динамического распределения свободной памяти. Первые три функции выделяют память, функция **free()** освобождает ее. Прототипы этих функций хранятся в заголовочных файлах ***stdlib.h*** и ***malloc.h***.

В таблице 1.1 представлены прототипы функций управления памятью и их краткое описание.

Таблица 1.1

Функции работы с динамической памятью

Название функции	*Прототип и краткое описание*
malloc	**void * malloc (unsigned s);** Возвращает указатель на начало области динамической памяти длиной в **s** байт. При неудачном завершении возвращает значение **NULL**
calloc	**void * calloc (unsigned n, unsigned m);** Возвращает указатель на начало области обнуленной динамической памяти, выделенной для размещения **n** элементов по **m** байт каждый. При неудачном завершении возвращает значение **NULL**
realloc	**void * realloc (void * bl, unsigned ns);** Изменяет размер блока ранее выделенной памяти до размера **ns** байт. **bl** – адрес начала изменяемого блока. Если **bl = NULL** (память раньше не выделялась), то функция выполняется как **malloc**
free	**void * free (void * bl);** Освобождает ранее выделенный участок динамической памяти, адрес первого байта которого равен значению **bl**

1.1.2 Указатели

Указатель является одной из наиболее важных концепций языка Си. Правильное понимание и использование указателей особенно необходимо по следующим причинам:

- при помощи указателей выполняется динамическое распределение памяти;
- указатели обеспечивают поддержку динамических структур данных (списки, деревья);
- указатели позволяют повысить эффективность программирования;
- с помощью указателей можно изменять значения передаваемых аргументов функций.

Для указателей определены понятия константы, переменной, массива. Как и любую переменную, указатель необходимо объявить. Объявление указателя состоит из имени базового типа, символа **`*`** и имени переменной. Тип указателя определяет тип объекта, на который указатель будет ссылаться.

Понятие указателя тесно связано с понятием адреса объекта. Операция, позволяющая получить адрес любой переменной, обозначается символом **`&`**. Операция косвенной адресации, обозначаемая символом **`*`**, называется еще операцией разыменования, или взятия значения по адресу.

Запись

```
n = *t;
```

означает, что переменной ***`n`*** присваивается значение, которое находится в ячейке памяти с адресом, записанным в переменную ***`t`***.

Операции над указателями

Над указателями определено пять основных операций в языке Си.

Определение адреса указателя: **`&p`**, где **`p`** – указатель.

Присваивание. Указателю можно присвоить адрес переменной ***`p = &q`***, где ***`p`*** – указатель, ***`q`*** – идентификатор переменной.

Определение значения, на которое ссылается указатель: ***`*p`*** (операция косвенной адресации или операция разыменования).

Увеличение (уменьшение) указателя. Увеличение выполняется как с помощью операции сложения с целым числом (**`+`**), так и с помощью операции инкремента (**`++`**). Эта операция дает предсказуемый результат только при работе с указателем на элемент массива. С помощью операции инкремента, например, указатель перемещается на следующий элемент массива. В общем случае указатель будет передвинут на размер ***`sizeof(type)`***, где ***`type`*** – базовый тип указателя. Уменьшение выполняется аналогично с помощью операции вычитания целого числа (**`–`**) либо декремента (**`--`**).

Например, пусть **p1** – указатель, ***sizeof*(*char*)** равно 1, ***sizeof*(*int*)** равно *n*, ***sizeof*(*float*)** равно 4, тогда **p1++** перемещает указатель на:

- 1 байт, если ***p1** имеет тип **char**;
- *n* байт, если ***p1** имеет тип ***int***;
- 4 байта, если ***p1** имеет тип ***float***.

Таким образом, операции адресной арифметики подчиняются следующим правилам. После увеличения значения переменной-указателя на единицу данный указатель будет ссылаться на следующий объект своего базового типа. После уменьшения – на предыдущий объект. Для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают. Указатель всегда ссылается на объект с типом, тождественным базовому типу указателя.

Разность двух указателей. Пусть **p1** и **p2** – указатели одного и того же типа. Операция вычитания двух указателей определяет количество объектов, расположенных между адресами, на которые указывают эти два указателя **p1** и **p2**. В частном случае, если **p1** и **p2** указывают на элементы массива, то их разность будет равна количеству элементов массива, расположенных между ними.

Сравнение двух указателей (**==**, **!=**). Обычно значение указателя сравнивается со значением **NULL**. В остальных случаях, как правило, сравнение указателей может оказаться полезным только тогда, когда два указателя ссылаются на общий объект.

Ниже приведен пример работы на Си с динамической памятью.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      float * t;
      int i, n;
      /* ввод числа: */
      printf("\nn=");
      scanf("%d",&n);
      /* выделение памяти для массива вещественных чисел: */
      t = (float *)malloc( n * sizeof(float) );
      if( t == NULL )
      {
            printf("Not enough memory");
            return -1;
      }
      /* память выделена успешно: */
      /* ввод массива вещественных чисел */
      for (i = 0; i < n; i++)
```

```
    {
        printf ("x[%d]= ", i);
        scanf("%f", &(t[i]));
    }
    /* печать массива вещественных чисел: */
    for ( i = 0; i < n; i++ )
    {
        if (i % 2 == 0)
            printf ("\n");
        printf("\tx[%d] = %f", i, t[i]);
    }
    free (t);  /* освобождение памяти */
    return 0;
}
```

Указатели и массивы

В языке Си [6, 8] массивы и указатели тесно связаны друг с другом. Например, когда объявляется массив в виде `int b[5]`, то при этом не только выделяется память для пяти элементов массива, но и формируется указатель с именем `b`, значение которого равно адресу нулевого элемента массива. Доступ к элементам массива может осуществляться через указатель с именем `b`. С точки зрения синтаксиса языка указатель `b` является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя.

Пример

Рассмотрим фрагмент программы работы с массивом.

```
int b[5] = {1, 1};
int * p, i;
for (i = 2; i < 5; i++)
     b[i] = b[i-1] + b[i-2];
/*-----------------*/
for (p = b+2; p != b+5; p++)
   *p = *(p-1) + *(p-2);
```

Первый цикл в примере обращается к элементам массива по индексу. Второй цикл производит те же самые действия, но работает с массивом посредством указателей. Оба фрагмента программы заполняют ячейки массива числами Фибоначчи.

Иллюстрация работы фрагмента приведена на рис. 1.2. Указатель `p` в начальный момент содержит адрес второй ячейки массива.

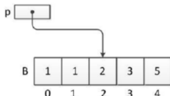


Рис. 1.2. Присваивание значений элементам массива со 2-го по 4-й

Указатели и строки

Символьную строку можно описать с помощью одномерного массива и с помощью указателя.

Описание символьной строки при помощи одномерного массива осуществляется следующим образом:

```
char name[n];
```

Например, для переменной **char name[16]** в памяти выделяется 16 байт, что дает возможность сформировать строку из 15 символов.

Инициализация возможна двумя способами:

– посимвольная инициализация

```
char name[10]={'h','e','l','l','o','\0'};
```

при этом оставшиеся четыре позиции не будут заполнены;

– инициализация на основе строковой константы

```
char name[10] = "Yes";
```

при этом в выделенную для строки память будут помещены три символа и добавлен четвертый символ **'\0'**.

В обоих случаях инициализация и объявление возможны без указания длины массива, например:

```
char name[]={'h','e','l','l','o','\0'};
```

В этом случае будет создан массив из шести элементов.

Описание символьной строки с помощью указателя:

```
char * s1;
```

здесь **s1** – переменная-указатель на тип **char**.

Однако заметим, что для инициализации указателя требуется задать необходимый для строки объем памяти.

Инициализация указателя на символьную строку может быть выполнена следующим образом:

– инициализация строковым литералом

```
char * st = "Hello";
```

– присваивание значение другого указателя

```
char * st = name;
```

где ***name*** – идентификатор массива или указатель на другую строку символов.

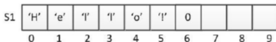
Примеры

```
char a[20]; // массив, в который можно записать строку
char * b;   // указатель, с которым можно связать строку
scanf ("%s", a); // считывание строки из входного потока
b = a;
```

Ниже приведены фрагменты программ работы со строками.

```
char * my_strcpy(char * dst, const char * src)
      // функция копирует байты строки src в строку dst
{
      char * p = dst;
      while (*src)
            *dst++ = *src++;
      *dst = 0;
      return p;
}
...
// описание и инициализация строк:
char s1[10]= "Hello!", s2[5]= "test";
...
my_strcpy (s1, s2);
```

Иллюстрация работы этого фрагмента показана на рис. 1.3 и 1.4. На рис. 1.3 показано начальное состояние памяти, а на рис. 1.4 – состояние памяти после копирования содержимого строки **S2** в строку **S1**.



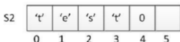
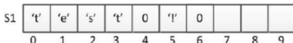


Рис. 1.3. Состояние памяти до копирования



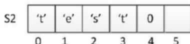


Рис. 1.4. Состояние памяти после копирования строки

В качестве примера опишем функцию, осуществляющую сравнение двух строк в лексикографическом порядке, и фрагмент программы, содержащий вызов этой функции.

```
int my_strcmp(const char * al, const char * ar)
     /* сравнивает строки al и ar;
          выдает значение:     > 0, если al > ar
                               < 0, если al < ar
                               = 0, если al равно ar */
{
     int i;
     for (i = 0; (al[i]==ar[i])&&(al[i]!=0); i++);
     return (al[i] - ar[i]);
}
...
char s1[10] = "Hello!";
char s2[5] = "test";
if (my_strcmp (s1, s2) > 0) // каков результат выражения?
```

Цикл в функции **`my_strcmp`** завершается при несовпадении очередных элементов строки либо при достижении конца строки одной из строк, в данном случае это строка **`al`**. Конец второй строки определяется при сравнении элементов строк. Если строки совпадают, то цикл будет завершен по второму неравенству.

Ответ на вопрос, заданный в последнем комментарии приведенного фрагмента программы, оставляем в качестве упражнения читателям.

1.1.3 Статические и динамические переменные

Подводя промежуточные итоги, отметим, что к *статическим переменным* относятся глобальные и локальные переменные программы, описанные в программе и обозначенные идентификаторами. Память под такие переменные отводится автоматически во время компиляции. Переменные, созданием и уничтожением которых может явно управлять программист, называются *динамическими переменными*. Такие переменные, количество которых и место расположения в памяти заранее не известно, невозможно обозначить идентификаторами. Поэтому единственным средством доступа к динамическим переменным является указатель на место их текущего расположения в памяти. Область памяти, в которой они располагаются, называется *кучей (heap)* или динамической памятью. Значения параметров сравнения статических и динамических переменных отражены в таблице 1.2.

Таблица 1.2

Сравнение статических и динамических переменных

Параметры сравнения	*Статические переменные*	*Динамические переменные*
Способ распределения памяти	Автоматическое (во время компиляции)	Управляется программой
Место расположения	Глобальные переменные и локальные **`static`** – в сегменте данных, локальные **`auto`** – на стеке	В динамической памяти (куче)
Способ доступа	По имени (идентификатор)	По адресу (указатель на место расположения в памяти)

Следует отметить, что локальные переменные в языке Си называются статическими, если объявлены с ключевым словом **`static`**, и в этом случае они располагаются в сегменте данных.

1.2 Списки

Список – это структура данных, представляющая собой конечную последовательность элементов, в которой каждый элемент посредством указателей связывается с другими элементами [4].

Таким образом, каждый элемент связанного списка хранит какую-либо информацию и связи с другими элементами. На рис. 1.5 показан элемент списка.

Рис. 1.5. Элемент списка

Списки позволяют представить набор объектов, однотипных или различных, которые удобно рассматривать как единое целое, например, «список студентов», «перечень документов», «опись деталей» и так далее. В то же время, всегда можно обратиться к любому элементу списка и обработать хранящуюся в нем информацию, при необходимости повторяя действия для каждого элемента списка.

Списки бывают односвязные, в том числе циклические, двусвязные, в том числе циклические, и иерархические.

1.2.1 Односвязные списки

Односвязный список – это список, у элементов которого существует связь, указывающая на следующий элемент списка. Элементами списка могут быть любые структуры. Первый элемент списка называется *головой* списка, остальная часть – *хвостом* (рис. 1.6).

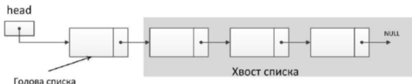


Рис. 1.6. Односвязный список

Описание односвязного списка, элементами которого являются строки, на языке Си представлено ниже.

```
typedef struct Node
{
      char word[256];       // область данных
      struct Node * next;   // указатель на следующий узел
} TList;

TList * new_el(const char * newWord)
{
      TList * p = (TList *)malloc(sizeof(TList));
      if( p )
      {
            strncpy(p->word, newWord, 256); //записать слово
            p->next = NULL;             // следующего узла нет
      }
      return p;       // результат функции - адрес элемента
}
```

Односвязные списки всегда линейны. Линейность односвязного списка вытекает из линейной логической упорядоченности его элементов, так как для каждого элемента, за исключением первого и последнего, имеются единственный предыдущий и единственный последующий элементы.

Создание первого элемента списка состоит в следующей последовательности описаний и действий:

```
TList * head = NULL; // голова списка
head = new_el("Hello");
```

Результат выполнения этой последовательности визуально представлен на рис. 1.7.

Рис. 1.7. Создание первого элемента списка и заполнение полей структуры **Tlist**

Существует несколько способов построения односвязного списка, а именно: помещать новые элементы в голову или в конец списка, вставлять элементы в определенные позиции списка. От способа построения списка зависит алгоритм функции добавления элемента.

Вставка элемента в голову односвязного списка реализуется с помощью описанной ниже функции **add_first()**.

```
TList * add_first (TList * head, const char * data)
{
	TList * cur = new_el(data);
	if(cur)
		cur->next = head;
	return cur;
}
```

Результат выполнения функции **add_first** показан на рис. 1.8.

```
head = add_first (head, "first");
```

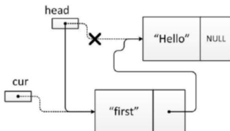


Рис. 1.8. Вставка элемента в голову односвязного списка

Вставка элемента в односвязный список после элемента с адресом **prev** выполняется с помощью функции **add_after()**.

```
void add_after (TList * prev, const char * data)
{
	TList * cur = new_el(data);
	if (cur)
	{
		cur->next = prev->next;
		prev->next = cur;
	}
}
```

Результат выполнения функции **add_after** показан на рис. 1.9.

```
add_after (after, "middle");
```

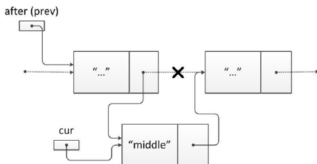


Рис. 1.9. Вставка элемента в односвязный список

Авторы предлагают выполнить вставку элемента в конец списка самостоятельно.

Последовательный перебор элементов односвязного списка осуществляется просто, начиная с головы списка и следуя указателям.

```
TList * p = head;
while (p)               // пока не дошли до конца списка
{
     p = p->next;       // переход к следующему
}
```

Функция **find(head, s)** возвращает указатель на элемент списка с головой **head**, содержащий строку **s**.

```
TList * find (TList * head, const char * s)
{
     TList * q = head;

     while (q && strcmp(q->word, s))
          q = q->next;
     return q;
}
```

Процедуру удаления элемента из односвязного списка легко выполнить, если известен адрес элемента, предшествующего удаляемому (**prev** на рис. 1.10).

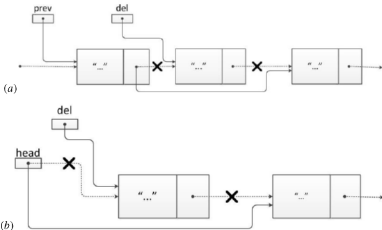


Рис. 1.10. Удаление элемента: (*a*) из середины односвязного списка,
(*b*) из начала односвязного списка

Предлагается реализовать эту процедуру самостоятельно, при этом не забыть освободить память, используя функцию **free**, которой необходимо передавать указатель на удаляемый элемент.

1.2.2 Двусвязные списки

Двусвязные списки – это списки, элементы которых имеют по две связи, указывающие на предыдущий и следующий элементы (см. рис. 1.11).

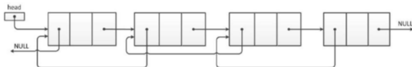


Рис. 1.11. Двусвязный список

Описание такого списка и создание первого элемента в нем показано ниже.

```
typedef struct Tlist2
{
     char word[256];           // область данных
     struct Tlist2 * next;     // ссылка на следующий узел
     struct Tlist2 * prev;     // ссылка на предыдущий узел
} TList2;
TList2 * head = NULL;          // голова списка
```

```
TList2 * new_el_2(const char * newWord)
{
      TList2 * p = (TList2 *)malloc(sizeof(TList2));
      if (p)
      {
            strncpy(p->word, newWord, 255); // записать слово
            p->next = NULL;             // следующего узла нет
            p->prev = NULL;             // предыдущего узла
      нет
      }
      return p;        // результат функции - адрес элемента
}
```

Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Во-первых, перемещение по списку возможно в обоих направлениях. Во-вторых, упрощается работа со списком, в частности, упрощается вставка и удаление элементов. Как и в односвязном списке, при вставке нового элемента в двусвязный список возможны три случая: элемент вставляется в начало, в середину и в конец списка. Эти операции показаны на рис. 1.12 и 1.13. Вставка элемента в двусвязный список после элемента с ненулевым адресом **pr** может быть выполнена следующим образом:

```
cur = new_el_2(data);
if (cur)
{
      cur->next = pr->next;
      if (pr->next)
            pr->next->prev = cur;
      pr->next = cur;
      cur->prev = pr;
}
```

Последовательность выполнения этих операторов приведет к результату, показанному на рис. 1.12. Этот же код подходит и к вставке в конец списка.

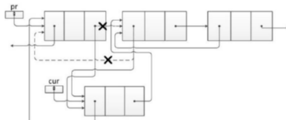


Рис. 1.12. Вставка элемента в двусвязный список

Удаление элемента с адресом **cur** может быть выполнено с помощью следующей последовательности операторов:

```
if (cur->prev)
     cur->prev->next = cur->next;
if (cur->next)
     cur->next->prev = cur->prev;
free(cur);
```

Результат выполнения этих операторов показан на рис. 1.13.

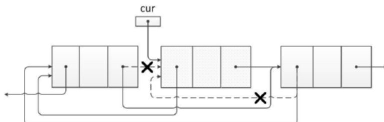


Рис. 1.13. Удаление элемента в двусвязном списке

Легко заметить, что приведенный вариант удаления применим к элементу, расположенному в любом месте двусвязного списка.

Вставку элемента в начало двусвязного списка предлагается реализовать самостоятельно.

1.2.3 Циклические списки

Линейные списки характерны тем, что в них можно выделить первый и последний элементы. Циклические списки также как и линейные бывают односвязными (рис. 1.14) и двусвязными. Основное отличие циклического списка состоит в том, что в списке нет нулевых указателей.

> *Циклический список* – это список, в котором связь последнего элемента указывает на первый или на один из других элементов этого списка.

Для полного обхода такого списка достаточно иметь указатель только на текущий элемент.

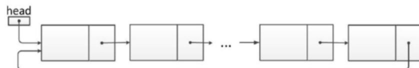


Рис. 1.14. Пример односвязного циклического списка

Двусвязный циклический список позволяет достаточно просто осуществлять операции вставки и удаления элементов слева и справа от текущего элемента. В отличие от линейного списка, элементы являются равноправными, и во многих случаях нет необходимости выделять первый элемент списка, достаточно иметь указатель на текущий элемент.

1.2.4 Иерархические списки

Иерархическими списками называются списки, значениями элементов которых являются указатели на другие списки (подсписки).

Пример такого списка показан на рис. 1.15.

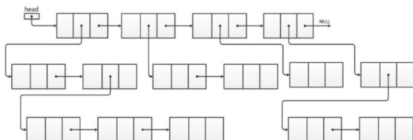


Рис. 1.15. Пример иерархического списка

К рассмотрению иерархических списков (ввод, вывод, создание) мы еще вернемся в главе 2.

1.3 Простые структуры данных

В общем случае *линейный список* – это множество, состоящее из n ($n \geq 0$) узлов (элементов) $X[1]$, $X[2]$, …, $X[n]$, структурные свойства которого ограничены линейным относительным положением узлов, т. е. следующими условиями:

если $n > 0$, то $X[1]$ – первый узел;

если $1 < k < n$, то k-му узлу $X[k]$ предшествует узел $X[k\text{-}1]$, а за узлом $X[k]$ следует узел $X[k+1]$;

$X[n]$ – последний узел.

Ниже перечислены наиболее популярные операции над линейными списками.

- Получить доступ к k-му элементу списка, проанализировать и/или изменить значения его полей.
- Включить новый узел перед k-м.
- Исключить k-й узел.
- Объединить два или более линейных списков в один.

- Разбить линейный список на два или более линейных списков.
- Сделать копию линейного списка.
- Определить количество узлов.
- Выполнить сортировку в возрастающем порядке по некоторым значениям полей в узлах.
- Найти в списке узел с заданным значением в некотором поле.

Заметим, что, как правило, не все операции нужны одновременно.

Выделяются виды линейных списков с ограниченным набором операций над ними, которые позволяют решать определенные классы задач. К таким видам относятся стеки и очереди, которые будут рассмотрены в следующих разделах.

1.3.1 Стеки

Стек – это линейный список, в котором все включения, исключения и всякий доступ выполняются на одном конце списка, называемой *вершиной стека*.

Схематично стек изображен на рис. 1.16.

Рис. 1.16. Стек

Часто в литературе можно прочесть, что стек – это структура данных, в которой доступ к элементам организован по принципу *LIFO* (*Last In – First Out*, «последним пришел – первым вышел»).

Добавление элемента в стек, называется также *проталкиванием* (**`push`**), и возможно только в вершину стека, таким образом, добавленный элемент становится первым сверху.

Удаление элемента, называется также *выталкиванием* (**pop**), и возможно только из вершины стека, при этом второй сверху элемент становится верхним.

Стек имеет следующие синонимы:

- *push-down* список,
- реверсивная память,
- гнездовая память,
- магазин,
- *LIFO* (*last-in-first-out*),
- список *йо-йо*.

В современных компьютерах стек используется для:

- размещения локальных переменных;
- размещения параметров процедуры или функции;
- сохранения адреса возврата из функции;
- временного хранения данных.

На стек выделяется ограниченная область памяти. При каждом вызове процедуры в стек добавляются новые элементы. Поэтому, если вложенных вызовов будет много, стек переполнится. Опасной в отношении переполнения стека является рекурсия.

При программировании стек чаще всего реализуется в виде односвязного списка. Каждый элемент структуры содержит указатель на следующий. Однако к этому виду списка по определению неприменима операция обхода элементов. Доступ возможен только к верхнему элементу структуры.

Возможны следующие операции работы со стеком.

makenull(S) – делает стек **S** пустым;
create() – создание стека, возвращает указатель на стек;
top(S) – выдает значение верхнего элемента стека, не удаляя его;
pop(S) – выдает значение верхнего элемента стека и удаляет его из стека;
push(S, x) – помещает в стек **S** новый элемент со значением **x**;
empty(S) – если стек пуст, то функция возвращает 1 (*истина*), иначе 0 (*ложь*).

Описание и реализация стека могут быть выполнены так же, как описание и реализация односвязного списка.

Далее будет показана другая реализация стека, где внутри структуры описывается указатель на верхушку стека. Это сделано для того, чтобы запретить доступ к другим элементам стека. В дальнейшем мы будем использовать именно эту реализацию.

```
typedef struct Stack_Item
{
     int data;
     struct Stack_Item * next;
} StackItem;
typedef struct _Stack
{
     StackItem * top;
} Stack;
```

Функция опустошения стека, создания стека и взятие верхнего элемента могут быть выполнены следующим образом.

```
void makenull (Stack * s)
{
     StackItem * p;
     while (s->top)
     {
          p = s->top;
          s->top = p->next;
          free(p);
     }
}

Stack * create ()
{
     Stack * s = (Stack *)malloc(sizeof(Stack));
     s->top = NULL;
     return s;
}

int top (Stack * s)
{
     if (s->top)
          return (s->top->data);
     else
          return 0; // здесь может быть реакция на ошибку –
                    // обращение к пустому стеку
}
```

В результате выполнения операции взятия верхнего элемента стека некоторой переменной *a* должно быть присвоено значение первого элемента стека, и значение указателя на начало списка должно быть перенесено на следующий элемент стека.

```
int pop(Stack * s)
{
     StackItem * p = s->top;
```

```
    int a = p->data;
    s->top = p->next;
    free(p);
    return a;
}
```

Занесение элемента в стек производится аналогично вставке нового элемента в начало списка с учетом добавления дополнительного указателя. Процедура занесения элемента в стек должна содержать два параметра: первый параметр содержит значение, которое нужно поместить в стек, а второй – указатель на стек.

```
void push(Stack * s, int a)
{
  StackItem * p = (StackItem *)malloc(sizeof(StackItem));
  if ( p )
  {
        p->data = a;
        p->next = s->top;
        s->top = p;
  }
}
```

Проверка на пустоту стека показана ниже.

```
int empty (Stack * s)
{
  return ( s->top == NULL );
}
```

1.3.2 Задача перевода выражения из инфиксной формы записи в постфиксную

Обычные арифметические выражения, содержащие скобки и используемые в повседневной практике, называют *инфиксными* выражениями [3]. Знак операции в них располагается между операндами. Порядок выполнения действий в таких выражениях определяется старшинством операций и скобками. Вычисление инфиксного выражения подразумевает его предварительный анализ с целью выявления порядка выполнения операций.

Существуют формы записи арифметических выражений без скобок, в которых порядок действий задается порядком знаков операций в выражении. Существует две формы записи: одна называется *префиксной*, вторая – *постфиксной* или *обратной польской записью* в честь их создателя – польского математика Яна Лукашевича (*Jan Lukasiewicz*). В префиксной форме знак операции предшествует операндам, в постфиксной знак операции следует за операндами.

Примеры

`a+(f-b*c/(z-x)+y)/(a*r-k)`	– инфиксная форма записи
`+a/+-f/*bc-zxy-*ark`	– префиксная форма записи
`afbc*zx-/-y+ar*k-/+`	– постфиксная форма записи

Особенности обратной польской (постфиксной) записи следующие. Порядок выполнения операций однозначно задается порядком следования знаков операций в выражении. В отличие от инфиксной записи, невозможно использовать одни и те же знаки для записи унарных и бинарных операций. Так, в инфиксной записи выражение `5 * (-3 + 8)` неверно записать как `5 3 - 8 + *`. И наконец, из-за отсутствия скобок обратная польская запись короче инфиксной. Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность ее операндов и ее знак, после чего выражение вычисляется дальше по тому же правилу. Результатом вычисления выражения становится результат последней вычисленной операции.

Вычисление бесскобочных выражений оказывается проще, чем выражений со скобками, поскольку операции должны выполняться в порядке описания и предварительный анализ не требуется. Рассмотрим алгоритм вычисления выражения, заданного в постфиксной форме. Этот алгоритм использует стек. Входными данными для алгоритма является строка, содержащая выражение, записанное в постфиксной форме. На выходе получаем число – значение заданного выражения. Операндами будем называть числа, строки, идентификаторы и т. п.

Алгоритм 1.1. Вычисление на стеке

Вход: строка – выражение, записанное в постфиксной форме.
Выход: число – значение заданного выражения.
Метод:

Шаг 0: Стек пуст.
Взять первый элемент из входной строки и поместить его в переменную X.

Шаг 1: Если X – операнд, то поместить его в стек.
Если X – знак операции, то вытолкнуть из стека два верхних элемента, применить к ним соответствующую операцию, результат положить в стек.

Шаг 2: Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе вытолкнуть из стека результат вычисления выражения.

Конец алгоритма 1.1.

Пример

Пусть дано следующее выражение: `5 2 3 * 4 2 / - 4 / +`

Ход и результат работы алгоритма 1.1 для данного примера представлен в таблице 1.3.

Таблица 1.3

Работа алгоритма 1.1 для выражения `5 2 3 * 4 2 / - 4 / +`

Вход	*Операция*	*Содержимое стека*
5	поместить в стек	5
2	поместить в стек	5, 2
3	поместить в стек	5, 2, 3
*	умножить два верхних элемента, результат положить в стек	5, 6
4	поместить в стек	5, 6, 4
2	поместить в стек	5, 6, 4, 2
/	применить деление к двум верхним элементам стека, результат положить в стек	5, 6, 2
-	применить вычитание к двум верхним элементам стека, результат положить в стек	5, 4
4	поместить в стек	5, 4, 4
/	применить деление к двум верхним элементам стека, результат положить в стек	5, 1
+	сложение двух верхних элементов стека, результат положить в стек	6

Этот алгоритм требует всего одного прохода выражения слева направо.

Перевод инфиксных выражений в постфиксные основан на использовании стека и таблицы приоритетов. Приоритеты операций указаны в таблице 1.4.

Таблица 1.4

Приоритеты операций

Знаки операций	*Приоритеты операций*
(	1
)	2
=	3
+, -	4
*, /	5

Алгоритм 1.2. Преобразование инфиксного выражения в постфиксное

Вход: строка – выражение, записанное в инфиксной форме.

Выход: строка – выражение, записанное в постфиксной форме.

Метод:

Шаг 0: Взять первый элемент из входной строки и поместить его в *X*. Выходная строка и стек пусты.

Шаг 1: Если *X* – операнд, то дописать его в конец выходной строки.
Если *X* = ′(′, то поместить его в стек.
Если *X* = ′)′, то вытолкнуть из стека и поместить в конец выходной строки все элементы до первой встреченной открывающей скобки. Эту скобку вытолкнуть из стека.
Если *X* – знак операции, отличный от скобок, то пока стек не пуст и верхний элемент стека имеет приоритет, больший либо равный приоритету *X*, выталкивать его из стека и помещать в выходную строку. Затем поместить *X* в стек.

Шаг 2: Если входная строка не исчерпана, то поместить в *X* очередной элемент входной строки и перейти на Шаг 1, иначе, пока стек не пуст, выталкивать из стека содержимое в выходную строку.

Конец алгоритма 1.2.

Пример

Пусть дана строка: `a+(f-b)*c/(z-x+y)`.

Ход работы алгоритма 1.2 с указанной входной строкой представлен в таблице 1.5.

Этот алгоритм также требует всего одного прохода по входной строке слева направо.

Таблица 1.5

Работа алгоритма перевода выражения из инфиксной формы в постфиксную

Ввод	*Содержимое выходной строки*	*Содержимое стека*
a	a	
+	a	+
(	a	+ (
f	a f	+ (
-	a f	+ (-
b	a f b	+ (-
)	a f b -	+
*	a f b -	+ *
c	a f b - c	+ *
/	a f b - c * +	/
(	a f b - c * +	/ (
z	a f b - c * + z	/ (
-	a f b - c * + z	/ (-
x	a f b - c * + z x	/ (-
+	a f b - c * + z x -	/ (+
y	a f b - c * + z x - y	/ (+
)	a f b - c * + z x - y +	/
	a f b - c * + z x - y + /	

1.3.3 Очереди

Очередь – это линейный список, в котором все включения производятся на одном конце списка, все исключения – на другом его конце (рис. 1.17).

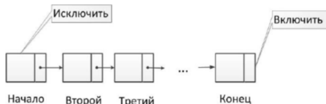


Рис. 1.17. Очередь

В очереди установлена следующая дисциплина доступа к элементам: «первый пришел – первый вышел» (*FIFO*, *First In – First Out*). Эту структуру данных можно представить в виде трубки. В один конец трубки добавляются шарики – элементы очереди, из другого конца они извлекаются. Элементы в середине очереди недоступны.

В принципе, можно было бы разрешить добавлять элементы в оба конца очереди и забирать их также из обоих концов. Такая структура данных в программировании тоже существует, ее название – *дек* (*Double Ended Queue*, т. е. очередь с двумя концами). Дек применяется значительно реже, чем очередь.

Очередь можно моделировать с помощью массива и с помощью списков. Как и при реализации стека, мы рассмотрим вариант моделирования очереди односвязным списком. Моделирование очереди посредством массива остается в качестве упражнения. При реализации очереди на массиве необходимо обратить внимание на то, что когда из очереди удаляется первый элемент, остальные элементы не должны сдвигаться, должен изменяться только индекс элемента массива, с которого начинается очередь. Чтобы не выйти за границы массива, можно замкнуть массив в кольцо: т. е. считать, что за последним элементом массива следует первый.

Ниже перечислены операции работы с очередью:

`makenull(q)` – делает очередь **`q`** пустой;
`create()` – создает очередь;
`first (q)` – выдает значение первого элемента очереди, не удаляя его;
`get(q)`
`dequeue(q)` – выдает значение первого элемента очереди и удаляет его из очереди;

put(q, x)
enqueue(q, x) – помещает в конец очереди **q** новый элемент со значением **x**;
empty (q) – если очередь пуста, то функция возвращает 1 (*истина*), иначе – 0 (*ложь*).

В виде односвязного списка очередь можно описать так, как показано ниже.

```
typedef struct Queue_Item
{
	int data;
	struct Queue_Item * next;
} QueueItem;
```

Аналогично описанию стека, опишем тип **Queue** (очередь), который будет содержать два поля – указатели на начало и на конец очереди.

```
typedef struct _Queue
{
	QueueItem *first;
	QueueItem *last;
} Queue;
```

Далее показана операция создания очереди:

```
Queue * create()
{
	Queue *q = (Queue *) malloc ( sizeof( Queue ) );
	q->first = NULL;
	q->last = NULL;
	return q;
}
```

Добавление элемента в очередь реализуется следующей функцией:

```
void put(Queue * q, int a)
{
  QueueItem * p = (QueueItem *)malloc(sizeof(QueueItem));
  p->data = a;
  p->next = NULL;
  if (q->first == NULL)
     q->first = p;
  else
     q->last->next = p;
  q->last = p;
}
```

Изъятие первого элемента из очереди:

```
int get(Queue * q)
{
	QueueItem * p = q->first;
	int a = p->data;
	q->first = p->next;
	free(p);
	if (q->first == NULL)
		q->last = NULL;
	return a;
}
```

Проверка очереди на пустоту:

```
int empty( Queue * q )
{
	return (q->first == NULL);
}
```

Пример

Рассмотрим представленный ниже фрагмент программы работы с очередью.

```
int main()
{
	Queue * q;
	int b;
	q = create();
	put(q, 5);
	put(q, 7);
	while (!empty(q))
	{
		b = get(q);
		printf("%d\n", b);
	}
	return 0;
}
```

В этой программе сначала очередь создается, затем в нее добавляются элементы, содержащие числа 5 и 7. В конце программы в цикле распечатывается содержимое очереди, которая в результате опустошается.

ГЛАВА 2. ГРАФЫ

Авторы предполагают, что читатель знаком с основными понятиями теории множеств. Напомним только некоторые из них.

По определению элементы множества считаются неупорядоченными. При некоторых обстоятельствах удобно рассматривать упорядоченные пары объектов. Поэтому дадим следующее определение.

Пусть a и b – объекты.

Через (a, b) обозначим *упорядоченную пару*, состоящую из объектов a и b, взятых в этом порядке.

Упорядоченные пары (a, b) и (c, d) называются *равными,* если $a = c$ и $b = d$.

В противоположность этому равенство множеств: $\{a, b\} = \{b, a\}$.

Декартовым произведением множеств A и B, обозначаемым через $A{\times}B$, называют множество $\{(a, b) \mid a \in A \text{ и } b \in B\}$.

Пример

Пусть $A = \{1, 2\}$ и $B = \{2, 3, 4\}$.

Тогда $A{\times}B = \{(1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)\}$.

2.1 Отношения

Многие распространенные математические понятия, такие как принадлежность, включение множеств, числовое неравенство, являются отношениями. Мы дадим формальное определение понятия отношения и посмотрим, как под это определение подходят известные примеры отношений.

Пусть A и B – множества. *Отношением из A в B* называется любое подмножество множества $A{\times}B$.

Если $A = B$, то говорят, что отношение *задано*, или *определено*, *на A* (или по-другому, что это – *отношение на* множестве A).

Если R – отношение из A в B и $(a, b) \in R$, то пишут aRb. Множество A называют *областью определения* отношения R, а множество B – *множеством его значений.*

Отношение $\{(b, a) \mid (a, b) \in R\}$ называется *обратным* к отношению R и часто обозначается через R^{-1}.

Понятие отношения очень общее. Часто отношение обладает рядом свойств, для которых установлены специальные названия.

Пусть A – множество и R – отношение на A.

Отношение R называется

– *рефлексивным*, если aRa для всех a из A;
– симметричным, если aRb влечет bRa для a и b из A;
– транзитивным, если aRb и bRc влекут aRc для a, b и c из A.

Элементы a, b и c не обязаны быть различными.

Отношение, обладающее свойствами рефлексивности, симметричности и транзитивности, называется *отношением эквивалентности.*

Важное свойство любого отношения эквивалентности R, определенного на множестве A, заключается в том, что оно разбивает множество A на непересекающиеся подмножества, называемые *классами эквивалентности.* Для каждого элемента $a \in A$ обозначим через $[a]$ класс эквивалентности, содержащий a, т. е. множество $\{b | aRb\}$.

Пример

Рассмотрим отношение сравнения по модулю N, определенное на множестве неотрицательных целых чисел. Говорят, что *a сравнимо с b по модулю N,* и пишут $a \equiv b\ (\ mod\ N\ )$, если существует такое целое число k, что $a - b = kN$. Пусть, например, $N = 3$. Тогда множество $\{0, 3, 6, ...\ 3n, ...\}$ будет одним из классов эквивалентности, поскольку $3n \equiv 3m\ (mod\ 3)$ для любых целых чисел m и n. Этот класс обозначим через [0]; можно его обозначить также через [3], или [6], или [3n], поскольку любой элемент класса эквивалентности является представителем этого класса.

Другие два класса эквивалентности отношения сравнения по модулю 3 таковы:

$[1] = \{1, 4, 7, ...\ 3n + 1, ...\}$

$[2] = \{2, 5, 8, ...\ 3n + 2, ...\}$

Объединение трех множеств [0], [1] и [2] совпадает с множеством всех неотрицательных целых чисел. Таким образом, это отношение эквивалентности разбивает множество всех неотрицательных целых чисел на три непересекающихся класса эквивалентности [0], [1] и [2] (рис. 2.1).

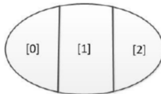


Рис. 2.1. Классы эквивалентности отношения сравнения по модулю 3

Индексом отношения эквивалентности, определенного на множестве A, называется число классов эквивалентности, на которые разбивается множество A этим отношением.

Важный класс отношений образуют отношения порядка. Вообще говоря, порядок на множестве A – это любое транзитивное отношение на A. При изучении алгоритмов важную роль играет специальный тип порядка,

называемый *частичным*. Множество, на котором задан какой-нибудь порядок, называется *упорядоченным.*

Частичным порядком на множестве A называется отношение R, определенное на A и такое, что

(1) R транзитивно,

(2) для всех $a \in A$ утверждение aRa ложно, т. е. отношение R *иррефлексивно.*

Из свойств (1) и (2) следует, что если aRb истинно, то bRa ложно. Это свойство называется *асимметричностью.*

В литературе частичным порядком иногда называют то, что мы называем рефлексивным частичным порядком.

Рефлексивным частичным порядком на A называется отношение R, обладающее следующими свойствами:

- R транзитивно,
- R рефлексивно,
- если aRb и bRa, то $a = b$.

Последнее свойство называют *антисимметричностью.*

В разделе 2.4 будет показано, что каждый частичный порядок можно представить графически в виде структуры, называемой ориентированным ациклическим графом.

Важный частный случай частичного порядка – линейный порядок.

Линейный порядок R на множестве A – это такой частичный порядок, что если a и b принадлежат A, то либо aRb, либо bRa, либо $a=b$.

Если A – конечное множество, то линейный порядок R удобно представлять себе, считая все элементы множества A расположенными в виде последовательности $a_1, a_2, \ldots a_n$, для которой a_iRa_j тогда и только тогда, когда $i < j$.

Аналогично можно определить рефлексивный линейный порядок, а именно, R – *рефлексивный линейный порядок* на A, если R – такой рефлексивный частичный порядок, что aRb или bRa для всех a *и* b из A.

Например, отношение $<$ (меньше), определенное на множестве неотрицательных целых чисел, является линейным порядком. Примером рефлексивного линейного порядка может служить отношение $\leq$ (меньше либо равно).

2.2 Основные определения

Неупорядоченный граф G – это пара (A, R), где A – множество элементов, называемых *вершинами* (или *узлами),* а R – отношение на множестве A.

Если R – несимметричное отношение, то G – *ориентированный граф*; если R – симметричное, то G – не*ориентированный граф.*

Пример

Пусть $A=\{1, 2, 3, 4\}$, $R = \{(1, 1), (1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (4, 3)\}$. Тогда можно изобразить этот граф $G = (A, R)$, занумеровав четыре точки числами 1, 2, 3, 4 и проведя стрелку из точки a в точку b, если $(a, b)\in R$ (рис. 2.2).

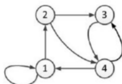


Рис.2.2. Пример ориентированного графа

Пара $(a, b)\in R$ в ориентированном графе называется *дугой* графа G, а в неориентированном – *ребром*.

На рис. 2.3. приведен пример такой дуги.

Рис. 2.3. Дуга в графе

Говорят, что дуга *выходит* из вершины a и *входит* в вершину b.

Если (a, b) – дуга, то говорят, что вершина a *предшествует* вершине b, а вершина b *следует за* вершиной a.

Если (a, b) – дуга, то говорят, что вершина b *смежна* с вершиной a.

Петлей называется ребро (дуга), начинающееся и заканчивающееся в одной вершине.

Последовательность вершин $(a_0, a_1, \ldots a_n)$, $n \geq 1$, называется *путем* (или *маршрутом*) *длины n* из вершины a_0 в вершину a_n, если для каждого $1 \leq i \leq n$ существует дуга, выходящая из вершины a_{i-1} и входящая в вершину a_i.

Простым путем (маршрутом) называется путь (маршрут) проходящий по каждой вершине ровно один раз.

В основном мы будем использовать введенное определение графа. Но для некоторых задач нам понадобится понятие мультиграфа.

Мультиграфы – это графы, в которых между двумя вершинами может быть более одного ребра (дуги). Такие ребра называются *кратными*.

Простым графом называется граф без петель и кратных ребер с конечным количеством вершин.

Пример

В графе на рис. 2.2. путем может быть следующая последовательность вершин: (1, 2, 4, 3).

Если существует путь из вершины a_0 в вершину a_n, то говорят, что a_n *достижима* из a_0.

Циклом называется путь (a_0, a_1, ... a_n), в котором $a_0 = a_n$.

Цикл, содержащий простой путь называется *простым циклом.*

Пример

В графе на рис. 2.2. последовательность (1, 2, 3, 4, 1) является циклом.

Ориентированный граф называется *сильно связным,* если для любых двух разных вершин *a и b* существует путь из *a* в *b*.

Неориентированный граф при выполнении этого свойства называется *связным*.

Степенью по входу (или *полустепенью входа*) вершины *a* назовем количество входящих в нее дуг, а *степенью по выходу* (или *полустепенью исхода*) – количество выходящих из нее дуг.

Если граф неориентированный, то *степень* вершины – это количество ребер, связанных с ней.

Пример

У вершины 2 в графе на рис. 2.2 полустепень входа равна 1, а полустепень исхода – 2.

Ациклическим графом называется (ориентированный) граф, не имеющий циклов.

В ациклических графах вершину, степень по входу которой равна 0, называют *базовой.*

Вершину, степень по выходу которой равна 0, называют *листом* (или *концевой* вершиной).

Пример

На рис. 2.4 в ациклическом графе базовыми вершинами являются вершины 1, 2, 3 и 4, а листами – вершины 2, 4, 7, 8 и 9.

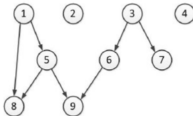


Рис. 2.4. Пример ациклического графа

Если (*a*, *b*) – дуга в ациклическом графе, то *a* – *прямой предок b*, а *b* – *прямой потомок* вершины *a*.

Если в ациклическом графе существует путь из a в b, то говорят, что a – *предок* b, а b – *потомок* вершины a.

Заметим, что если R – частичный порядок на множестве A, то (A, R) – ациклический граф. Более того, если (A, R) – ациклический граф и R' – отношение «являться потомком», определенное на A, то R' – частичный порядок на A.

2.3 Представление графов в памяти машины

Пусть дан граф $G = (V, E)$. Обозначим через $N = |V|$ – количество вершин в графе, через $M = |E|$ – количество дуг.

На примере графа, показанного на рис. 2.5, рассмотрим основные способы представления графов в оперативной памяти. Здесь вершины пронумерованы числами от 1 до 4, а дуги – от 1 до 6.

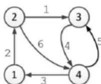


Рис. 2.5. Пример графа

Все введенные в этом разделе определения даны для ориентированных графов. Но их можно использовать и для неориентированных графов.

2.3.1 Матрица смежности

Матрица смежности для графа G – это матрица A размера $N{\times}N$, состоящая из 0 и 1, в которой $A[i, j] = 1$ тогда и только тогда, когда есть дуга из вершины i в вершину j.

Матрица смежности для графа, показанного на рис. 2.5, представлена ниже на рис. 2.6.

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	1	0

Рис. 2.6. Матрица смежности для графа на рис. 2.5

2.3.2 Матрица инцидентности

Матрица инцидентности для графа G – это матрица B размера $N \times M$, в которой:

$$B[i, j] = \begin{cases} 1, \text{ если дуга } j \text{ инцидентна вершине } i, \\ -1, \text{ если дуга } j \text{ входит в вершину } i, \\ 0, \text{ если дуга } j \text{ не связана с вершиной } i. \end{cases}$$

Матрица инцидентности для графа, показанного на рис. 2.5, представлена ниже на рис. 2.7. Строки матрицы соответствуют вершинам, а столбцы – дугам.

	1	2	3	4	5	6
1	0	1	–1	0	0	0
2	1	–1	0	0	0	1
3	–1	0	0	1	–1	0
4	0	0	1	–1	1	–1

Рис. 2.7. Матрица инцидентности для графа на рис. 2.5

2.3.3 Списки смежности

Списком смежности для вершины v называется список всех вершин w, смежных с v.

Пример

Пусть дан граф, изображенный на рис. 2.8.

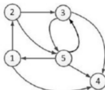


Рис. 2.8. Пример графа

Тогда этот граф можно представить в виде списка или массива вершин (рис. 2.9), каждый элемент которого является указателем на список смежности, соответствующий этой вершине.

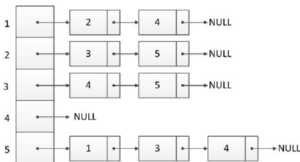


Рис. 2.9. Представление графа с рис. 2.8. в виде списков смежности

2.3.4 Табличное представление списков смежностей

Для графа на рис. 2.8 табличное представление списков смежности будет выглядеть, как показано на рис. 2.10. В первой колонке этой таблицы указывается вершина графа, во второй колонке – номер строки, в которой записана следующая смежная вершина для вершины из первой колонки.

	Номер вершины	Следующий
1	1	6
2	2	8
3	3	10
4	4	0
5	5	12
6	2	7
7	4	0
8	3	9
9	5	0
10	4	11
11	5	0
12	1	13
13	3	14
14	4	0

Рис. 2.10. Табличное представление списков смежности графа на рис. 2.8

Например, для вершины 1 в строке 6 записана смежная с ней вершина 2, а следующая смежная вершина 4 записана в строке 7.

2.4 Топологическая сортировка графа

Часто возникает необходимость преобразовать частичный порядок R на множестве A в линейный порядок, содержащий этот частичный порядок.

Проблема вложения частичного порядка в линейный называется *топологической сортировкой.*

Заметим, что если R – частичный порядок на множестве A, то (A, R) – ациклический граф. Более того, если (A, R) – ациклический граф и R' – отношение «являться потомком», определенное на A, то R' – частичный порядок на A.

Топологическая сортировка является одним из основных алгоритмов на графах, который применяется для решения множества более сложных задач. Она применяется в самых разных ситуациях, например:

- когда решение большой задачи разбивается на ряд подзадач, над которыми установлен частичный порядок: без решения одной задачи нельзя решить несколько других;
- при определении последовательности чтения курсов в учебных программах: один курс основывается на другом;
- при составлении расписания выполнения работ на одном станке: одну работу следует выполнить раньше другой.
- при создании карты сайта, если при описании разделов используется древовидная система.

Интуитивно топологическая сортировка заключается в том, что надо взять ациклический граф, который, в сущности, и является заданным частичным порядком, и расположить его вершины в ряд так, чтобы все дуги были направлены в одну сторону. Или, другими словами, требуется *перенумеровать* вершины графа таким образом, чтобы каждое ребро вело из вершины с меньшим номером в вершину с большим номером.

Линейный порядок задается положением вершин в этом ряду. Например, для графа, показанного на рис. 2.4, линейный порядок его вершин представлен на рис 2.11.

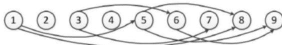


Рис. 2.11. Топологическая сортировка вершин графа, показанного на рис. 2.4.

Формально можно сказать, что частичный порядок R на множестве A *вложен в линейный порядок R',* если R' – линейный порядок и $R \subseteq R'$, т. е. aRb влечет $aR'b$ для всех a и b из A.

Топологическая сортировка может быть не единственной.

Алгоритм 2.1. Топологическая сортировка

Вход: Частичный порядок R на конечном множестве A.

Выход: Линейный порядок R' на A, *для* которого $R \subseteq R'$.

Метод: Так как A – конечное множество, линейный порядок R' на множестве A можно представить в виде списка $a_1, a_2, ..., a_n$, для которого $a_i\ R'\ a_j$ выполняется, если $i < j$, и $A = \{a_1, a_2, ..., a_n\}$. Эта последовательность элементов строится с помощью следующих шагов:

Шаг 1. Положить $i = 1$, $A_i = A$ и $R_i = R$.

Шаг 2. Если A_i пусто, то остановиться и выдать $a_1, ... a_i$, в качестве искомого линейного порядка. В противном случае выбрать в A_i такой элемент a_{i+1} что $a'\ R\ a_{i+1}$, ложно для всех $a' \in A_i$.

Шаг 3. Положить $A_{i+1} = A_i \setminus \{a_{i+1}\}$ и $R_{i+1} = R_i \cap (A_{i+1} \times A_{i+1})$. Затем увеличить i на единицу и перейти на Шаг 2.

Конец алгоритма 2.1

Если частичный порядок представлен в виде ациклического графа, то алгоритм 2.1 допускает особенно простую интерпретацию. На каждом шаге алгоритма пара (A_i, R_i) является ациклическим графом и a_{i+1} – его базовая вершина. Ациклический граф (A_{i+1}, R_{i+1}) образуется из (A_i, R_i) вычеркиванием вершины a_{i+1} и всех выходящих из нее дуг.

Пример

Пусть $A = \{a, b, c, d\}$ и $R = \{(a, b), (a, c), (b, d), (c, d)\}$. Так как a – единственная вершина, для которой $a'Ra$ ложно при всех $a' \in A$, надо взять $a_1 = a$.

Тогда $A_2 = \{b, c, d\}$ и $R_2 = \{(c, d), (b, d)\}$. Теперь в качестве a_2 можно взять либо b, либо c. Выберем $a_2 = b$. Тогда $A_3 = \{c, d\}$ и $R_3 = \{(c, d)\}$. Продолжая аналогично, найдем $a_3 = c$ и $a_4 = d$.

В результате получился линейный порядок:

$$R' = \{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}.$$

2.4.1 Реализация алгоритма на матрице смежности

Если граф хранить в памяти в виде матрицы смежности A, то действия алгоритма 2.1 сведутся к следующей последовательности шагов.

- Найти вершину, в которую не входит ни одна дуга. В матрице смежности ей соответствует столбец с одними нулями.
- Записать эту вершину в конец формируемого списка вершин.
- Удалить все выходящие из нее дуги: в матрице смежности обнулить соответствующую строку.
- Пока не перебрали все вершины, повторять описанную последовательность действий.

2.4.2 Реализация алгоритма на иерархических списках

Рассмотрим реализацию алгоритма 2.1 при условии, что граф задается в виде иерархического списка. Рассмотрим для примера граф G с множеством вершин `{1, 2, 3, 4, 5, 6}`, задающий частичный порядок `1<2; 2<5; 4<1; 2<6; 3<1`. Этот граф может быть представлен в виде иерархического списка на рис. 2.12.

Частичный порядок R задается во входном файле в виде последовательности пар вершин. Первая вершина пары предшествует второй вершине этой пары. Для нашего примера во входном файле будет задана последовательность пар: `(1, 2), (2, 5), (4, 1), (2, 6), (3 1)`.

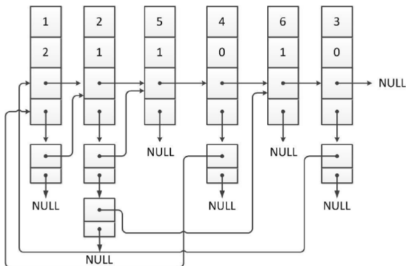


Рис. 2.12. Представление графа в виде иерархического списка

Элемент списка вершин графа состоит из четырех полей (рис. 2.13). В поле *ключ* хранится номер вершины, в поле *счетчик* – количество входящих дуг в данную вершину, в поле *следующий* – указатель на следующую вершину графа, заданную во входном файле. Поле *потомок* содержит указатель на список смежных вершин данной вершины.

Рис. 2.13. Структура элементов списка

На языке Си описание вершины и смежной для нее вершины выглядит следующим образом:

```
typedef struct t_node
{
      char key;
      int count;
      struct t_node *next;
      struct t_adjacent *trail;
} node;

typedef struct t_adjacent
{
      struct t_node *id;
      struct t_adjacent *next;
} adjacent;
```

На первом этапе построим иерархический список. Для этого будем считывать пары вершин из входного файла, и если вершина новая, т. е. она еще не была нами рассмотрена, будем добавлять ее в список, и после этого установим связи между созданными вершинами.

Для нашего примера первой будет рассмотрена вершина 1, для нее создадим элемент списка. В поле *ключ* запишем эту вершину. В поле *счетчик* запишем 0, так как входящих дуг пока нет. В поля *следующий* и *потомок* запишем **NULL**. Затем считываем из входного файла следующую вершину 2. Эта вершина новая, создадим для нее аналогичным образом элемент списка. Теперь можно установить связь между вершинами 1 и 2.

А именно, в поле *следующий* вершины 1 поместим указатель на вершину 2, создадим элемент списка для потомка вершины 1 и в поле *потомок* этой вершины установим указатель на вершину 2. Таким образом, мы получим список для двух первых вершин (рис. 2.14).

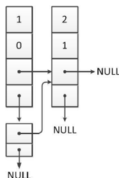


Рис. 2.14. Начало списка

Далее продолжим процесс аналогичным образом для следующих вершин из входного файла.

На втором этапе будет выполнена топологическая сортировка. Для этой цели список будет перестроен следующим образом. Сначала найдем в списке все вершины, значение поля *счетчик* которых равно нулю. После этого просмотрим элементы списка и уменьшим значения полей *счетчик* у потомков этих вершин на единицу, что равносильно удалению исходящих ребер в графе. Если в итоге поле *счетчик* у потомка стал равен нулю, то добавляем эту вершину в список после обрабатываемой.

Для нашего примера найдем первую вершину, в поле *счетчик* которой записан 0. Это вершина 4, у нее нет входных дуг. Затем в начало списка добавится вершина 3, в которую тоже не входит ни одна дуга. Потомком вершины 3 является вершина 1. Уменьшим значение поля *счетчик* вершины 1 на единицу, оно станет равным единице. Перейдем к следующей по списку вершине со значением поля *счетчик* равным нулю, это вершина 4. Потомком вершины 4 является вершина 1. Уменьшение значения *счетчика* вершины 1 приведет к тому, что это значение станет равным нулю. Следовательно, мы можем добавить ее в перестроенный список после вершины 4. Будем повторять этот процесс до тех пор, пока не обработаем все элементы иерархического списка. В результате получим список, представленный на рис. 2.15. Последовательность вершин в нем задает искомую топологическую сортировку.

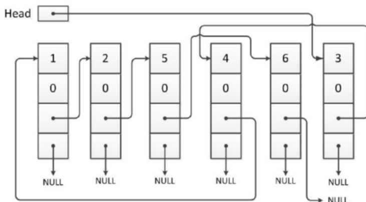


Рис. 2.15. Перестроенный список

Вершины будут выстроены в следующем порядке:

```
3, 4, 1, 2, 5, 6.
```

2.5 Обход графа в глубину

В данном разделе рассматривается метод поиска в глубину в графе (*Depth-first search, DFS*) [5].

Пусть задан граф $G = (V, E)$. Алгоритм поиска описывается следующим образом: для каждой непройденной вершины необходимо найти все непройденные смежные вершины и повторить поиск для них. Иными словами, это способ обхода вершин графа, который, начавшись от какой-либо вершины, рекурсивно применяется ко всем вершинам, в которые можно попасть из текущей.

Изложим реализацию алгоритма так, как описано в [5]. Будем различать *белые*, *серые* и *черные* вершины**.** Белая вершина – это вершина, которая еще не обнаружена при поиске. В процессе поиска вершина при ее обнаружении становится серой. Вершина становится черной, когда она обработана, то есть когда список смежных с ней вершин полностью просмотрен. Дополнительно алгоритм поиска в глубину использует метки времени. Каждая вершина имеет две метки. $d[u]$ отражает время, когда вершина впервые обнаружена и стала серой, а $f[u]$ – время, когда закончена обработка смежных с ней вершин и вершина окрашивается в черный цвет. Для меток времени в алгоритме *DFS* используется глобальная переменная текущего времени *time*. Текущий цвет произвольной вершины *v* будем хранить в элементе массива *цвет*[*v*]**.** Также для каждой вершины необходимо хранить вершину, откуда мы к ней пришли в процессе поиска в глубину. Для этой цели

используется метка *отец*[*v*]. Множество смежных вершин вершины *v* будем обозначать как *смежные*(*v*).

Алгоритм 2.2. Обход графа в глубину

Вход: Граф $G = (V, E)$, все вершины окрашены в белый цвет.
Выход: Граф *G* с вершинами, окрашенными в черный цвет.
Метод:

```
начало
    time ← 0
    для всех вершин u: u ∈ V выполнить
        цвет[ u ] ← белый
        отец[ u ] ← NULL
    конец цикла
    для всех вершин u: u ∈ V выполнить
        если цвет[ u ] = белый то
            Поиск( u )
    конец цикла
конец
процедура Поиск ( u )
    цвет [ u ] ← серый
    d[ u ] ← time
    time ← time + 1
    для всех v: v ∈ смежные(u) выполнить
        если цвет[ v ] = белый то
            начало
                отец [ v ] ← u
                Поиск( v )
            конец
    конец цикла
    цвет[ u ] ← черный
    f[ u ] ← time
    time ← time + 1
конец процедуры
```

Конец алгоритма 2.2

В процедуре *Поиск* выполняются следующие действия. Цвет текущей вершины становится *серым*, устанавливается время входа в нее, осуществляется проверка смежных с ней вершин на предмет, были ли они уже рассмотрены, т. е. окрашены в серый или черный цвет. Если смежная вершина еще не была рассмотрена, то ее предком считается текущая вершина, и для нее осуществляется вызов процедуры *Поиск*. После рассмотрения всех

смежных вершин текущая вершина окрашивается в черный цвет и устанавливается время окончания ее обработки.

Поиск в глубину работает за время, пропорциональное сумме количества вершин и количества ребер в графе G. Действительно, общее число операций при выполнении основного цикла алгоритма 2.2 оценивается как $O(|V|)$, где $|V|$ – количество вершин. При условии, что граф хранится в виде списков смежных вершин или матрицы смежности, общее число операций при выполнении процедуры *Поиск*(u) оценивается следующим образом. Цикл выполняется | *смежные* (v) | раз. Сумма всех смежных вершин для текущей вершины v не превышает количества всех ребер $|E|$ в графе G. Таким образом, общее число операций – $O(|V|+|E|)$.

Пример

На рисунке 2.16 показан обход в глубину, начинающийся с вершины 1.

Из вершины 4 есть ребро, ведущее в вершину 1, но эта вершина уже была посещена, поэтому в нее идти нельзя. Так как больше смежных непосещенных вершин у вершины 4 нет, то мы возвращаемся в вершину 2, откуда по ребру (2, 5) переходим в вершину 5.

По ребру (1, 2) переходим в вершину 2, затем по ребру (2, 4) – в вершину 4.

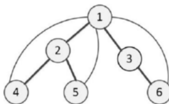


Рис. 2.16 Обход графа в глубину

У вершины 5 нет непосещенных смежных вершин, поэтому мы возвращаемся в вершину 2, а затем в вершину 1 по ребрам (5, 2) и (2, 1). Из вершины 1 переходим в вершину 3, а из нее в вершину 6. Дальше идти некуда, возвращаемся по ребрам (6, 3) и (3, 1) обратно в вершину 1, где заканчиваем обход.

Поиск в глубину на неориентированном графе $G = (V, E)$ разбивает ребра, составляющие E, на два множества T и B.

Ребро (v, w) помещается в множество T, если вершина w не посещалась до того момента, когда мы, рассматривая ребро (u, w), оказались в вершине v. В противном случае ребро (v, w) помещается в множество B.

Ребра из T будем называть *древесными,* а из B – *обратными.*

Подграф (V, T) представляет собой неориентированный лес, называемый *остовным лесом* для G, построенным поиском в глубину, или, короче, *глубинным остовным лесом* для G.

Заметим, что если граф связен, то глубинный остовный лес будет деревом. Вершина, с которой начинался поиск, считается корнем соответствующего дерева.

Подграф (V, T) также называют *подграфом предшествования*, так как множество древесных ребер T можно еще представить следующим образом:

$T = \{ (отец[v], v) : v \in V$ и $отец[v] \neq NULL \}$

2.5.1 Свойства поиска в глубину

Поиск в глубину – важная стратегия обхода графа, поскольку используется во многих различных приложениях. При поиске в глубину, если в окрестности вершины x, окрашенной в серый цвет, обнаруживается новая вершина y, то она помещается в стек и при следующем повторении цикла будет также окрашена в серый цвет. При этом x остается в стеке и через какое-то время снова станет активной. Иначе говоря, ребра, инцидентные вершине x, будут исследованы не подряд, а с перерывами.

Обозначим момент начала обработки вершины **`x`** – окрашивание в серый цвет – через **`"(x"`**, а окончание ее обработки – окрашивание в черный цвет – через **`"x)"`**. Тогда время начала и окончания обработки вершины при поиске в глубину образуют правильную скобочную структуру.

Теорема 2.1

При поиске в глубину в графе $G = (V, E)$ для любых двух вершин **`u`** и **`v`** выполняется одно из следующих утверждений:

- Отрезки **`[d[u],f[u]]`** и **`[d[v],f[v]]`** не пересекаются.
- Отрезок **`[d[u],f[u]]`** целиком содержится внутри отрезка **`[d[v],f[v]]`** и **`u`** является потомком **`v`** в дереве поиска в глубину.
- Отрезок **`[d[v],f[v]]`** целиком содержится внутри отрезка **`[d[u],f[u]]`** и **`v`** является потомком **`u`** в дереве поиска в глубину.

Без доказательства

Пример

Рассмотрим дерево, построенное обходом в глубину, представленное на рис. 2.17.

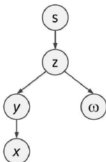


Рис. 2.17. Дерево обхода графа в глубину

Ниже представлены значения массивов **d** и **f** и скобочная структура, соответствующие этому дереву обхода:

```
 1  2  3  4 5  6  7 8  9  10
(s (z (y (x x) y)(w w) z) s)
```

2.5.2 Поиск в глубину в ориентированном графе

Рассмотрим ориентированный граф из [3], изображенный на рис. 2.18(*a*). На рис. 2.18(*b*) показан обход графа в глубину, начиная с вершины v_1. В результате обхода графа в глубину получаем подграф предшествования, который в общем случае представляет собой глубинный остовный лес.

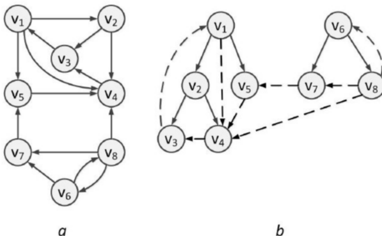


Рис. 2.18. *b* – глубинный остовный лес для графа *a*

Поиск в глубину в ориентированном графе G разбивает множество его дуг на четыре класса:

1) *древесные дуги,* идущие к новым вершинам в процессе поиска;
2) *прямые дуги,* идущие от предков к подлинным потомкам, но не являющиеся древесными ребрами;
3) *обратные дуги,* идущие от потомков к предкам, возможно, из вершины в себя;
4) *поперечные дуги,* соединяющие вершины, которые не являются ни предками, ни потомками друг друга.

Теорема 2.2

Вершина v является потомком вершины u в лесу поиска в глубину тогда и только тогда, когда в момент $d[u]$ обнаружения вершины u существует путь из u в v, состоящий только из белых вершин.
Без доказательства

Теорема 2.3

Дуга **(u, v)** является

а) *прямой дугой* тогда и только тогда, когда
d[u] < d[v] < f[v] < f[u];

б) *обратной дугой* тогда и только тогда, когда
d[v] < d[u] < f[u] < f[v];

в) *поперечной дугой* тогда и только тогда, когда
d[v] < f[v] < d[u] < f[u].

Без доказательства

2.5.3 Решение задачи топологической сортировки методом поиска в глубину

Решение задачи топологической сортировки может быть выполнено с помощью метода обхода в глубину. Для этого создается список просмотренных вершин следующим образом. Выполняется процедура, аналогичная процедуре *Поиск* из алгоритма 2.2, которая здесь называется *Топологическая сортировка,* от текущей вершины, а далее обработанная вершина заносится в начало строящегося списка. Далее представлено описание алгоритма топологической сортировки.

Алгоритм 2.3. Топологическая сортировка (методом поиска в глубину)

Вход: Ациклический граф $G = (V, E)$, все вершины окрашены в белый цвет. Список вершин S пуст.

Выход: Список S, содержащий все вершины графа G, построенные в линейном порядке, включающем исходный частичный порядок.

Метод:

начало

$S \leftarrow \varnothing$

для всех вершин u: $u \in V$ **выполнить**

цвет[u] ← белый

отец[u] ← NULL

конец цикла

для всех вершин u: $u \in V$ **выполнить**

если *цвет*[u] = *белый* **то**

Топологическая сортировка(u)

конец цикла

конец

процедура *Топологическая сортировка* (u)

цвет [u] ← *серый*

для всех $v \in$ *смежные*(u) **выполнить**

если *цвет*[v] = *белый* **то**

начало

отец [v] ← u

Топологическая сортировка(v)

конец

конец цикла

цвет[u] ← *черный*

Поместить вершину u в начало списка S

конец процедуры

Конец алгоритма 2.3

Пример

Для ациклического графа на рис. 2.4, список, содержащий линейный порядок вершин, полученный обходом в глубину, представлен на рис. 2.19. Вершины на каждом шаге алгоритма выбирались в порядке их нумерации.

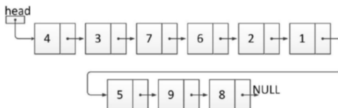


Рис. 2.19. Топологическая сортировка графа с рис. 2.4, полученная методом поиска в глубину

2.6 Обход графа в ширину

Поиск в ширину (*breadth-first search*) – один из базовых алгоритмов, лежащий в основе многих других алгоритмов для графов.

Пусть задан граф $G = (V, E)$ и начальная вершина s. Алгоритм поиска в ширину [5] перечисляет все достижимые из s вершины в порядке возрастания расстояния от s. Расстоянием считается количество ребер кратчайшего пути. В результате из графа выделяется часть, называемая «деревом поиска в ширину» с корнем s. В этом дереве путь из корня s в любую вершину будет одним из кратчайших путей из s в графе. Алгоритм называется поиском в ширину, так как поиск идет вширь: сначала выбираются все смежные вершины для заданной, затем смежные смежных и т. д.

2.6.1 Метод поиска в ширину

Для наглядности так же, как и при поиске в глубину, будем использовать раскраску вершин [5]. *Белая* вершина – вершина, которая еще не обнаружена при поиске. *Серая* – это вершина, уже обнаруженная при поиске в качестве смежной, но у которой еще не просмотрены ее смежные вершины. *Черная* – вершина, обнаруженная при поиске, и у нее просмотрены все ее смежные вершины. В ходе поиска обнаруженные вершины поочередно становятся серыми и в дальнейшем – черными. Пусть в начальный момент времени все вершины окрашены в белый цвет.

Вершину s окрасим в серый цвет и припишем ей расстояние 0. Смежные с ней вершины окрасим в серый цвет, припишем им расстояние 1, их предок – это вершина s. Окрасим вершину s в черный цвет.

На шаге n поочередно рассматриваем белые вершины, смежные с вершинами с пометками n–1, и каждую из них раскрашиваем в серый цвет, приписываем им предка и расстояние n. После этого вершины с расстоянием n–1 окрашиваются в черный цвет.

В качестве промежуточной структуры хранения при обходе в ширину используется очередь. Ниже описан алгоритм поиска в ширину. Здесь и далее в пособии функции работы с очередью обозначены так же, как в п. 1.3.3.

Алгоритм 2.4. Поиск в ширину

Вход: Неориентированный связный граф $G = (V, E)$, вершина s – начало обхода, очередь Q пуста.

Выход: Дерево (V, T) обхода графа G в ширину с корнем в вершине s, которое задается подграфом предшествования. Для каждой вершины $v \in \mathbf{V}$, исключая s, ребро $(\textit{отец}[v], v) \in T$, $d[v]$ – длина кратчайшего пути от s до v.

Метод:

```
1   для всех вершин u ∈ ( V \ {s} ) выполнить
2       цвет [u] ← белый
3       отец[u] ← NULL
4       d[u]← ∞
5   конец цикла
6   цвет [s] ← серый
7   d[s] ← 0
8   отец[s] ← NULL
9   Q ← create()
10  put (Q, s)
11  пока (empty (Q) = ложь) выполнить
12      u ← first (Q)
13      для всех вершин v: v ∈ смежные( u ) выполнить
14          если цвет[v] = белый то
15              начало
16                  цвет [v]← серый
17                  отец[v] ← u
18                  d[v] ← d[u] + 1
19                  put(Q, v)
20              конец
21      конец цикла
22      get(Q)
23      цвет[u] ← черный
    конец цикла
```

Конец алгоритма 2.4

Итак, вначале дерево обхода графа в ширину состоит только из корня вершины s. Вершина s окрашивается в серый цвет и помещается в очередь. Расстояние до нее устанавливается равным 0.

На каждом витке основного цикла алгоритма обрабатываем первую в очереди вершину. Она, как и все вершины в очереди, окрашена в серый цвет. Мы просматриваем все смежные с ней белые вершины, окрашиваем их в серый цвет и помещаем их в очередь, установив для них длину пути на единицу большую, чем до вершины, которая находится в обработке. После этого перекрашиваем эту вершину в черный цвет и извлекаем из очереди. Так продолжаем до тех пор, пока очередь не будет пуста.

Алгоритм 2.4 правильно работает и для ориентированных графов.

Обозначим через $\delta(s,v)$ – минимальную длину пути из вершины s в v.

Лемма 2.1

Пусть $G = (V, E)$ – ориентированный или неориентированный граф, s – произвольная вершина из V. Тогда для любого ребра $(u, v) \in E$ справедливо соотношение $\delta(s, v) \le \delta(s, u) + 1$.

Доказательство. Если вершина u достижима от вершины s за k шагов, то и v достижима от s не более чем за $k + 1$ шаг.

Лемма 2.2

Если $\delta(s, v) > 0$, то существует вершина u, до которой кратчайшее расстояние от s на единицу меньше ($\delta(s, v) = \delta(s, u) + 1$) и для которой v является смежной.

Доказательство. Рассмотрим кратчайший путь из s в v. Его длина $\delta(s, v)$. Возьмем вершину u, лежащую на этом пути непосредственно перед v. Убедимся, что до нее расстояние на единицу меньше. У нас есть ведущий в нее путь длины $\delta(s, v) - 1$. Более короткого пути не может быть по лемме 2.1.

Теорема 2.4

Для любого целого неотрицательного k существует момент после нескольких повторений тела цикла алгоритма 2.4, когда выполнены следующие утверждения:

- вершины, для которых расстояния от начальной вершины меньше k – черные, ровно k – серые, больше k – белые;
- в очереди Q находятся серые вершины, и только они;
- в массиве d хранятся правильные значения расстояний для черных и серых вершин, и бесконечные значения для белых;
- если v – серая или черная вершина, то $\delta(s, \textit{отец}[v]) = \delta(s, v) - 1$, и в графе есть ребро $(\textit{отец}[v], v)$; для белых вершин значение *отец* есть NULL.

Доказательство ведется индукцией по k. Докажем первые три утверждения. После выполнения строк 1–10 все пункты теоремы выполнены для $k = 0$.

Пусть для некоторого k утверждения выполнены. После нескольких итераций цикла из очереди будут забираться вершины (*просматриваемые*). Для смежных с ними белых вершин будут выполняться строки 16-19 – они добавляются в конец очереди (*добавляемые*).

В какой-то момент из очереди будут изъяты все вершины с расстоянием k и останутся только вновь добавленные. В этот момент мысленно прервем выполнение алгоритма 2.4 и убедимся, что все условия для $k + 1$ выполнены. Рассмотрим два из них.

Просматриваемые вершины по индукции находятся на расстоянии k. Добавляемые вершины находятся на расстоянии $k + 1$, так как они смежны

с просматриваемыми, находящимися на расстоянии k и по лемме 2.1 расстояние до них $\le k + 1$. Однако добавляются только белые вершины, по индукции до них расстояние > k.

Будут добавлены все вершины, находящиеся на расстоянии $k + 1$, так как если вершина v находится на расстоянии $k+1$, то по лемме 2.2 существует вершина u на расстоянии k, для которой она смежна, а так как она просматриваемая, то v добавится в очередь.

Остальные утверждения предлагается проверить самостоятельно.

Заметим, что при обходе в глубину, чем позднее будет посещена вершина, тем раньше она будет использована, поскольку просмотренные вершины накапливаются в стеке. При обходе графа в ширину используется очередь, поэтому, чем раньше посещается вершина, тем раньше она удаляется из очереди.

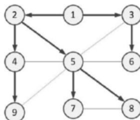


Рис. 2.20. Прямые ребра графа при обходе в ширину выделены жирными стрелками

Для графа, показанного на рис. 2.20 порядок обхода вершин методом поиска в ширину, начиная с вершины 1, задается следующей последовательностью: 1, 2, 3, 4, 5, 6, 9, 7, 8 **.** Можно также получить дерево обхода в ширину, если отмечать каждую прямое ребро, как это сделано на рис. 2.20.

Ниже представлен фрагмент алгоритма вывода кратчайших путей из вершины s в вершину v.

Алгоритм 2.5. Вывод пути

Вход: граф *G*, вершины *s* и *v*.
Выход: кратчайший путь из вершины *s* в вершину *v*.
Метод: вызов рекурсивной процедуры *Печать_пути* (*G*, *s*, ***v***)

```
процедура Печать_пути( G, s, v )
    если (s = v) то
        печать ( s )
    иначе
        если (отец[ v ] = NULL) то
            печать ("Нет пути")
```

иначе

Печать_пути (G, s, *отец*[v])

печать (v)

конец процедуры

Конец алгоритма 2.5

В алгоритме 2.5 каждая вершина помещается в очередь не более одного раза. Таким образом, для операций с очередью требуется $O(|V|)$ времени. Для каждой вершины просматриваются списки смежности, сумма длин всех просмотренных списков равна числу ребер $|E|$, на их обработку требуется $O(|E|)$ времени. Инициализация алгоритма обхода в ширину требует $O(|V|)$ шагов. Таким образом, время работы алгоритма равно $O(|V|+|E|)$.

2.6.2 Нахождение кратчайшего пути в лабиринте

Задача поиска выхода из лабиринта известна с древних времен. В терминах графов ее постановка следующая: лабиринт – это неориентированный граф, вершины которого представляют точки ветвления лабиринта, а ребра – пути между соседними точками. Одна или несколько вершин отмечены как выходы. Задача состоит в построении пути из некоторой исходной вершины в вершину-выход.

Часто лабиринт задается прямоугольной матрицей размера $n \times m$, и, например, нули в матрице обозначают проход, минус единицы – стены. Нужно найти путь из заданной клетки в другую заданную клетку, являющуюся выходом.

Алгоритм 2.6. Поиск кратчайшего пути в лабиринте

Вход: Карта лабиринта, которая представляет собой матрицу размера $n \times m$ с отмеченными проходами и стенками, координаты входной и выходной клеток.

Выход: Карта лабиринта с отмеченным кратчайшим путем из входной клетки в выходную.

Метод:

Шаг 1. Пометить входную клетку числом 1 и поместить ее в очередь.

Шаг 2. Взять из очереди клетку. Если это выходная клетка, то перейти на шаг 4, иначе пометить все непомеченные соседние клетки числом, на 1 большим, чем рассматриваемая, и поместить их в очередь.

Шаг 3. Если очередь пуста, то выдать «*Выхода нет*» и выйти, иначе перейти на шаг 2.

Шаг 4. *Конец обхода лабиринта*: если требуется вывести длину пути, то выдать число, которым помечена выходная клетка и выйти.

Шаг 5. *Обратный ход* (выполняется, если требуется определить не только длину пути, но и сам путь): начиная с выходной клетки,

каждый раз смещаться на клетку, помеченную на единицу меньшим числом, чем текущая клетка, пока не дойдем до входной клетки. При проходе выделять клетки, составляющие кратчайший путь, некоторым определенным числом.

Конец алгоритма 2.6

Пример

Пусть задан лабиринт размером 10×10 клеток, показанный на рис. 2.21. Стрелки указывают на входную и выходную клетки. Заштрихованные клетки – стены, а пустые клетки – проходы.

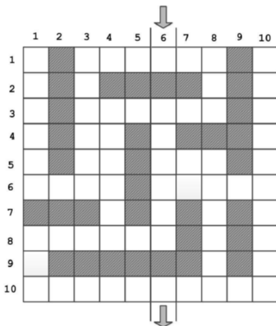


Рис. 2.21. Исходный лабиринт

Состояние памяти после применения к лабиринту алгоритма обхода в ширину показано на рис. 2.22. Числа в клетках обозначают номер хода, на котором впервые попадаешь в соответствующую клетку. Кратчайший путь от входа к выходу обозначен темным цветом.

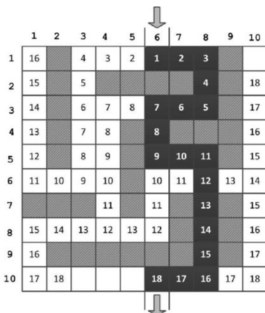


Рис. 2.22. Лабиринт с помеченными клетками в порядке его обхода в ширину и выделенным кратчайшим путем

2.7 Раскраска графов

В приложениях теории графов нередко возникают задачи, для решения которых требуется разбить множество всех вершин графа на несколько непересекающихся подмножеств таким образом, чтобы вершины в каждом подмножестве были попарно не смежными, а число таких подмножеств – минимально возможным. При решении таких задач можно раскрашивать вершины графа в различные цвета таким образом, чтобы любые две смежные вершины были раскрашены в разные цвета, а число использованных цветов было минимально возможным.

Заметим, что раскрашивать можно как вершины, так и ребра графа. Коснемся сначала задачи о раскраске вершин, будем считать при этом, что граф неориентированный и не является мультиграфом.

Пусть $G = (V, E)$ – неориентированный граф и k – натуральное число.

Функция $f : V \to \{1, \dots, k\}$ называется *раскраской* графа.

Раскраска называется *правильной*, если для любых смежных вершин $x, y \in V$ справедливо неравенство $f(x) \neq f(y)$. Число k – *количество красок* раскраски f.

Раскраску вершин графа можно рассматривать как разбиение множества вершин $V=V_1 \cup V_2 \cup V_3 ... \cup V_k$, где V_i – множество вершин цвета i.

Наименьшее число красок, необходимое для правильной раскраски графа G, называется *хроматическим числом* графа G.

Правильную раскраску таким числом красок будем называть *оптимальной*. Хроматическое число обозначается через $\chi(G)$.

Заметим, что если граф является *полным*, т. е. *любые две вершины являются смежными*, то хроматическое число такого графа равно n, где n – количество вершин.

Пример

На рис. 2.23 показаны оптимальные раскраски для графов.

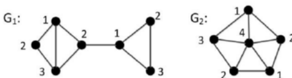


Рис. 2.23. Примеры раскрашенных графов, где $\chi(G_1) = 3$, $\chi(G_2) = 4$

Практическое применение алгоритмов раскраски графов демонстрируют приведенные ниже задачи.

2.7.1 Задача составления расписаний

Предположим, что в учебном центре надо провести несколько занятий за кратчайшее время. Длительность всех занятий одинакова. Некоторые занятия не могут проводиться одновременно, например, это занятия в одной и той же учебной группе (по разным предметам), или занятия проводит один и тот же преподаватель.

Для решения этой задачи построим граф G, вершинам которого взаимно-однозначно соответствуют занятия. Две вершины соединены ребром, если соответствующие занятия нельзя проводить одновременно.

Ясно, что правильная раскраска графа G определяет расписание, удовлетворяющее требованиям несовместимости по времени: занятия, соответствующие вершинам, окрашенным одинаково, можно проводить одновременно.

Справедливо и обратное, любое такое расписание определяет правильную раскраску графа G. Следовательно, кратчайшее время, необходимое для проведения всех занятий, равно $\chi(G)$, а из оптимальной раскраски графа G получается необходимое расписание.

2.7.2 Задача распределения ресурсов

Требуется выполнить некоторое множество работ $V = \{v_1, v_2, \ldots, v_n\}$. Имеется множество ресурсов $S=\{s_1, s_2, \ldots, s_r\}$, необходимых для выполнения этих работ. При этом:

- каждая работа использует некоторую часть указанных ресурсов;
- одновременно могут выполняться работы, использующие разные ресурсы;
- для каждой из работ требуется для выполнения одно и то же время t.

Нужно распределить ресурсы так, чтобы общее время выполнения всех работ было минимальным.

Для решения задачи построим граф G с множеством вершин V, соответствующим работам. Две различные вершины v и v' графа G смежны тогда и только тогда, когда для выполнения работ v и v' требуется хотя бы один общий ресурс.

Тогда наименьшее время выполнения всех работ будет равно $\chi(G)\cdot t$, а оптимальная раскраска графа G будет определять распределение ресурсов между работами.

2.7.3 Задача экономии памяти

Предположим, что необходимо написать программу для вычисления функции $\varphi(x_1, x_2, \ldots x_n)$, показанной на рис. 2.24.

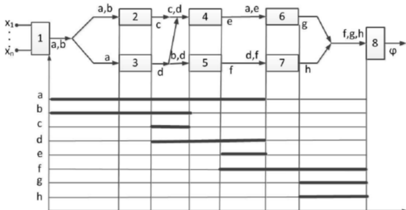


Рис. 2.24. Пример функции с указанием времен жизни переменных

Вычисление этой функции разбито на ряд блоков, у каждого из блоков имеются входные и выходные переменные. Время жизни каждой переменной изображено жирной линией. Будем считать, что значения каждой переменной занимают одну ячейку памяти.

Задача состоит в том, чтобы определить наименьшее число ячеек памяти, необходимое для хранения указанных в блок-схеме переменных.

Решить эту задачу можно следующим образом. На множестве переменных $V = \{a, b, c, d, e, f, g, h\}$ введем структуру графа: две переменных соединим ребром, если времена их жизни пересекаются. Полученный граф будем называть графом несовместимости переменных (рис. 2.25).

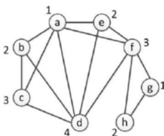


Рис. 2.25. Раскрашенный граф для функции с рис. 2.24

Значения переменных не могут занимать одну ячейку памяти тогда и только тогда, когда переменные соединены ребром в графе несовместимости. Следовательно, задача экономии памяти сводится к нахождению оптимальной раскраски графа несовместимости.

2.7.4 Алгоритм последовательной раскраски

К сожалению, задача раскраски произвольного графа минимальным количеством цветов принадлежит классу задач, которые называются *NP-полными* задачами. Это означает, что для раскраски вершин графа сначала пытаются использовать один цвет, затем – два цвета, потом – три и т. д., пока не будет получена подходящая раскраска вершин. В некоторых частных случаях можно предложить более быстрое решение, но в общем случае, это полный перебор возможных вариантов.

Изменим постановку задачи, будем искать не оптимальное решение, а близкое к нему. Отказавшись от требования минимального количества цветов раскраски графа, можно построить алгоритмы раскраски, которые работают значительно быстрее, чем алгоритмы полного перебора. Иногда эти эвристические алгоритмы дают оптимальный результат. Рассмотрим один из них – алгоритм последовательной раскраски графа.

Алгоритм 2.7. Последовательная раскраска графа

Вход: Неориентированный граф $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$.
Выход: Правильная раскраска графа G $f : V \to \{1, \ldots, k\}$.

Обозначения:

смежные(v) – список вершин, смежных с вершиной v;

цвет [v] – номер краски, приписанной вершине v; равен 0, если вершина еще не окрашена;

занят[i] – вспомогательный массив, в котором на i-м месте стоит 1, если цвет с номером i использован для окрашивания какой-то вершины, и стоит 0, если ни одна вершина не окрашена этим цветом.

Метод:

```
Шаг 0:  Упорядочить вершины графа G: V={v1, v2,…, vn}
        для всех v: v ∈ V выполнить
            цвет [v] ← 0
        конец цикла
        k ← 1
        i ← 1
Шаг 1:  цвет[vi] ← k
        i ← i + 1
        если i > n то
            перейти на Шаг 2
        для всех j от 1 до n выполнить
            занят [j] ← 0
        конец цикла
        для всех v: v ∈ смежные( vi ) и цвет[ v ] > 0 выполнить
            занят[ [цвет [v] ] ← 1
        конец цикла
        k ← 1
        пока занят [k] ≠ 0 выполнить
            k ← k + 1
        конец цикла
        перейти на Шаг 1
Шаг 2 : k ← цвет[v1]
        i ← 2
        пока i ≤ n выполнить
            если цвет[vi ] > k то
                k ← цвет[vi]
            i ← i + 1
        конец цикла
```

Конец алгоритма 2.7

2.7.5 Проблема четырех красок

Графы с хроматическим числом 2 – это такие графы, у которых множество вершин можно разбить на два независимых множества, в каждом из которых все вершины не имеют смежных вершин, входящих в это множество.

Для графов с *хроматическим числом* 3 такого простого описания неизвестно. Неизвестны и простые алгоритмы, проверяющие, можно ли данный граф раскрасить в три цвета.

Проблема четырех красок возникла в математике в середине XIX в. Первоначально вопрос формулировался так: сколько нужно красок для такой раскраски любой географической карты, при которой соседние страны раскрашены в разные цвета?

Достаточно ли четырех красок для раскраски любой географической карты?

Для ответа на этот вопрос нужно решить, достаточно ли четырех красок, чтобы раскрасить любой *планарный* граф – граф, в котором можно так расположить вершины, что ребра, соединяющие их, не пересекаются.

Эта проблема вызвала большой интерес в математике. Есть свидетельства, что ей занимались известные математики А. Ф. Мебиус и О. де Морган. В 1880 г. А. Компе опубликовал положительное решение проблемы четырех красок. Однако в 1890 г. Р. Хивуд обнаружил ошибку в этом доказательстве. Он доказал, что любой планарный граф можно раскрасить пятью красками.

После этого появлялось довольно много «доказательств» гипотезы четырех красок и «контрпримеров» к ней, в которых обнаруживались ошибки.

В 1969 г. Х. Хели свел проблему четырех красок к исследованию множества C так называемых неустранимых конфигураций. Множество C является конечным, но довольно большим (порядка нескольких тысяч).

Несколькими годами позже, в 1976 г. математикам К. Аппелю и В. Хейкену удалось показать, что все конфигурации из множества C можно правильно раскрасить в четыре цвета. В возникающем при этом переборе существенно использовался компьютер.

Такое решение проблемы четырех красок долгое время не признавалось многими математиками, поскольку его сложно повторить. Однако сейчас практически общепризнано, что К. Аппелем и В. Хейкеном доказана гипотеза четырех красок.

2.7.6 Раскраска ребер

Наряду с задачей о раскраске вершин имеется задача о *раскраске ребер* графа, когда цвета назначаются ребрам.

Раскраска ребер (или реберная раскраска) называется правильной, если любые два ребра, имеющие общую вершину, окрашены в разные цвета.

Минимальное число цветов, необходимое для правильной раскраски ребер графа G, называется *хроматическим индексом* графа и обозначается через $\chi'(G)$.

Обозначим через $\Delta(G)$ максимальную степень вершины в графе. При правильной реберной раскраске все ребра, инцидентные одной вершине, должны иметь разные цвета. Отсюда следует, что для любого графа выполняется неравенство $\chi'(G) \geq \Delta(G)$. Для некоторых графов имеет место строгое неравенство. Теорема, доказанная В. Г. Визингом в 1964 г., показывает, что $\chi'(G)$ может отличаться от $\Delta(G)$ не более чем на 1.

ГЛАВА 3. ДЕРЕВЬЯ

(*Ориентированным*) *деревом* T называется (ориентированный) граф $G = (A, R)$ со специальной вершиной $r \in A$, называемой корнем, у которого:

– степень по входу вершины r равна 0,
– степень по входу всех остальных вершин дерева T равна 1,
– каждая вершина $a \in A$ достижима из r.

Дерево T обладает следующими свойствами:

– T – ациклический граф,
– для каждой вершины дерева T существует единственный путь, ведущий из корня в эту вершину.

Поддеревом дерева $T = (A, R)$ называется любое дерево $T' = (A', R')$, у которого:

– A' не пусто и содержится в A,
– $R' = (A' \times A') \cap R$,
– ни одна вершина из множества $A \setminus A'$ не является потомком вершины из множества A'.

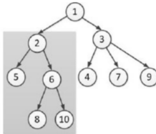


Рис. 3.1. Дерево и поддерево

На рис. 3.1 представлено дерево с корнем в вершине 1. Примером его поддерева может служить дерево с корнем в вершине 2. Все вершины, достижимые из вершины 2, входят в выделенное поддерево.

Ориентированный граф, состоящий из нескольких деревьев, называется *лесом*.

Пусть $T = (A, R)$ – дерево, $(a, b) \in R$ – дуга в дереве T, тогда говорят, что вершина a – *отец* b, а вершина b – *сын* a.

Глубина или *уровень вершины* – это длина пути от корня до этой вершины.

Высота вершины – длина максимального пути от этой вершины до листа.

Высота дерева – длина максимального пути от корня до листа.

Глубина корня равна 0.

Неориентированным деревом называется неориентированный ациклический связный граф.

Корневое неориентированное дерево – это неориентированное дерево, в котором одна вершина выделена в качестве корня.

Ориентированное дерево можно превратить в неориентированное, объявив все его ребра неориентированными. Мы будем употреблять одну и ту же терминологию, и пользоваться одними и теми же обозначениями для корневых неориентированных и для ориентированных деревьев. Основное различие здесь состоит в том, что все пути в ориентированном дереве идут от предков к потомкам, тогда как в корневом неориентированном дереве пути могут идти в обоих направлениях.

3.1 Бинарные деревья

Упорядоченное дерево – это дерево, в котором множество сыновей каждой вершины упорядочено слева направо.

Бинарное дерево – это упорядоченное дерево, в котором:
- любой сын – либо левый, либо правый,
- любая вершина имеет не более одного левого и не более одного правого сына.

На рис. 3.2 приведен пример бинарного дерева. Направление относительно расположения вершины любой из выходящих из нее дуг показывает, к какому сыну ведет дуга – левому или правому.

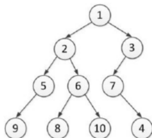


Рис. 3.2. Бинарное дерево

Бинарное дерево называется *полным*, если существует некоторое целое k, такое что любая вершина глубины меньше k имеет как левого, так и правого сына, а если вершина имеет глубину k, то она является листом.

Пример полного бинарного дерева высоты 3 представлен на рис. 3.3. В данном примере $k = 3$.

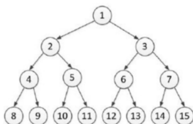


Рис. 3.3. Пример полного бинарного дерева

Бинарное дерево называется *почти полным*, если существует неотрицательное целое k такое, что:

- каждый лист в дереве имеет уровень k или $k + 1$;
- если вершина дерева имеет правого потомка уровня $k + 1$, тогда все его левые потомки, являющиеся листами, также имеют уровень $k + 1$.

Далее описано, как можно хранить полное или почти полное бинарное дерево в виде массива.

Пусть $T[2^{k+1} - 1]$ – массив для хранения вершин дерева, k – высота дерева. В $T[0]$ хранится корень дерева. Левый сын вершины i расположен в позиции $2 \cdot i+1$, правый сын – в позиции $(2 \cdot i + 2)$. Отец вершины, находящейся в позиции $i > 1$, расположен в позиции $\lfloor (i - 1) / 2 \rfloor$.

3.2 Обходы деревьев

Обход дерева – это способ методичного исследования вершин дерева, при котором каждая вершина просматривается только один раз.

В данном разделе будут рассмотрены обходы деревьев *в глубину* и *в ширину*.

3.2.1 Обходы деревьев в глубину

Пусть T – дерево, r – корень, $v_1, v_2, \ldots, v_n$ – сыновья вершины r.

Прямой (*префиксный*) обход:

- посетить корень r;
- посетить в прямом порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$.

Обратный (*постфиксный*) обход:

- посетить в обратном порядке поддеревья с корнями $v_1, v_2, \ldots, v_n$;
- посетить корень r.

Внутренний (*инфиксный*) обход (применяется только для бинарных деревьев):

- посетить во внутреннем порядке левое поддерево корня *r* (если существует);
- посетить корень *r;*
- посетить во внутреннем порядке правое поддерево корня *r* (если существует).

На рис. 3.4 приведены примеры прямого, обратного и внутреннего обходов одного и того же дерева. Вершины пронумерованы в порядке соответствующего обхода.

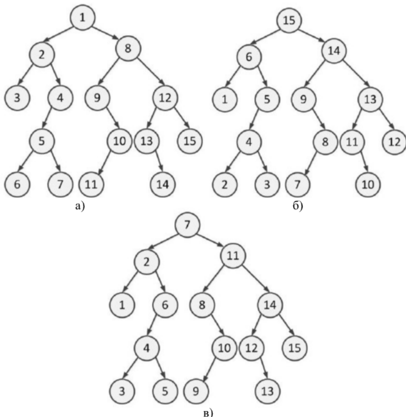


Рис. 3.4. Примеры обходов дерева: а) прямой, б) обратный, в) внутренний

На рис. 3.5 приведен пример дерева-формулы, вершинами которого являются знаки операций и идентификаторы, причем идентификаторы расположены только в листьях дерева.

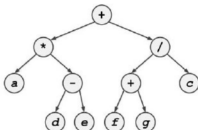


Рис. 3.5. Дерево-формула

При обходе вершин дерева разными порядками и распечатывании их содержимого получим следующие представления выражений:

`+ * a - d e / + f g c` – при префиксном обходе,
`a d e - * f g + c / +` – при постфиксном обходе,
`a * (d - e)+ (f + g) / c` – при инфиксном обходе.

Таким образом, в результате префиксного обхода строится выражение в префиксной форме, в результате постфиксного обхода – выражение в постфиксной форме, в результате инфиксного обхода – выражение в инфиксной форме. При этом в последнем случае для сохранения порядка операций используются скобки.

3.2.2 Обход деревьев в ширину

Обход дерева в ширину – это обход вершин дерева по уровням, начиная от корня, слева направо (или справа налево).

Алгоритм 3.1. Обход дерева в ширину слева направо (справа налево)

Вход: T – дерево, r – его корень.
Выход: Последовательность вершин дерева в порядке обхода в ширину
Метод:

Шаг 0: $Q \leftarrow \varnothing$. Очередь Q – пуста.
Поместить в очередь Q корень дерева r.

Шаг 1: Взять из Q очередную вершину, обработать ее.
Поместить в очередь Q всех сыновей этой вершины по порядку слева направо (справа налево).

Шаг 2: Если очередь Q пуста, то конец обхода, иначе перейти на Шаг 1.

Конец алгоритма 3.1

Пример

На рис. 3.6 представлено дерево, которое нужно обойти в ширину и распечатать ее вершины в порядке обхода. Работа алгоритма для данного примера представлена в таблице 3.1. В правом столбце таблицы 3.1 показаны вершины дерева в порядке их обхода.

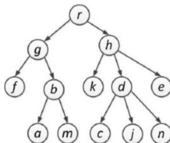


Рис. 3.6. Пример дерева для обхода в ширину

Таблица 3.1

Порядок обхода дерева (рис. 3.6) в ширину

Номер шага	*Действие*	*Содержимое очереди*	*Вывод вершины*
0	Поместить в очередь корень	`r`	
1	Взять из очереди первую вершину ***r***		`r`
	Поместить в очередь вершины ***g*** и ***h***	`g h`	
2	Взять из очереди первую вершину ***g***	`h`	`g`
	Поместить в очередь вершины ***f*** и ***b***	`h f b`	
3	Взять из очереди первую вершину ***h***	`f b`	`h`
	Поместить в очередь вершины ***k***, ***d*** и ***e***	`f b k d e`	
4	Взять из очереди первую вершину ***f***	`b k d e`	`f`
5	Взять из очереди первую вершину ***b***	`k d e`	`b`
	Поместить в очередь вершины ***a*** и ***m***	`k d e a m`	
6	Взять из очереди первую вершину ***k***	`d e a m`	`k`
7	Взять из очереди первую вершину ***d***	`e a m`	`d`
	Поместить в очередь вершины ***c***, ***j*** и ***n***	`e a m c j n`	
8	Взять из очереди первую вершину ***e***	`a m c j n`	`e`
9	Взять из очереди первую вершину ***a***	`m c j n`	`a`
10	Взять из очереди первую вершину ***m***	`c j n`	`m`
11	Взять из очереди первую вершину ***c***	`j n`	`c`
12	Взять из очереди первую вершину ***j***	`n`	`j`
13	Взять из очереди первую вершину ***n***		`n`

3.3 Представление деревьев

Дерево является двумерной структурой, но во многих ситуациях удобно пользоваться лишь одномерными структурами данных. Поэтому, мы заинтересованы в том, чтобы иметь для деревьев одномерные представления, сохраняющие всю информацию, которая содержится в двумерных картинках. Под этим мы подразумеваем, что двумерную картинку можно воспроизвести по ее одномерному представлению.

Очевидно, что одно из одномерных представлений дерева $T = (A, R)$ – это запись самих множеств A и R.

Существуют и другие представления, которые мы рассмотрим в этом разделе.

3.3.1 Скобочные представления деревьев

Левое скобочное представление дерева T (обозначается $Lrep\ (T)$) можно получить, применяя к нему следующие рекурсивные правила:

- Если корнем дерева T служит вершина a с прямыми потомками, которые являются корнями поддеревьев $T_1, T_2, \dots, T_n$, расположенными в этом порядке, то:
$$Lrep\ (T) = a\ (Lrep\ (T_1), Lrep\ (T_2), \dots, Lrep\ (T_n))$$
- Если корнем дерева T служит вершина a, не имеющая прямых потомков, то:
$$Lrep\ (T) = a.$$

Правое скобочное представление дерева T (обозначается $Rrep\ (T)$) описывается аналогично:

- Если корнем дерева T служит вершина a с прямыми потомками, которые являются корнями поддеревьев $T_1, T_2, \dots, T_n$, расположенными в этом порядке, то:
$$Rrep\ (T) = (Rrep\ (T_1), Rrep\ (T_2), \dots, Rrep\ (T_n))\ a.$$
- Если корнем дерева T служит вершина a, не имеющая прямых потомков, то
$$Rrep\ (T) = a.$$

Заметим, что левое скобочное представление соответствует прямому обходу дерева, а правое – обратному.

Пример

Для дерева T на рис. 3.6 имеют место следующие левое и правое скобочные представления:

$Lrep\ (T) = r\ (\ g\ (\ f, b\ (\ a, m\ )\ ), h\ (\ k, d\ (\ c,\ j, n\ ), e\ )\ )$

$Rrep\ (T) = (\ (\ f, (\ a, m\ )\ b\ )\ g, (\ k, (\ c,\ j, n\ )\ d, e\ )\ h\ )\ r$

3.3.2 Представление дерева списком прямых предков

Пусть дерево имеет n вершин, и вершины дерева пронумерованы числами от 1 до n. Список прямых предков составляется следующим образом. Для вершин дерева с номерами 1, 2, ... , n в этом же порядке перечисляются их прямые предки. Будем считать, что предком корня является 0.

Пример

Рассмотрим дерево на рис. 3.7, и построим для него список прямых предков. Для этого создадим массив, индексы которого изменяются от 1 до n, где n – количество вершин в дереве. Вершины дерева уже пронумерованы числами от 1 до n в произвольном порядке. Заполним наш массив числами по следующему правилу: в ячейку с номером i записывается номер вершины, которая является прямым предком вершины с номером i.

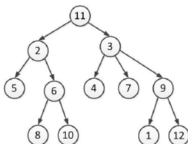


Рис. 3.7. Дерево с нумерованными вершинами

Полученный массив будет содержать нужный нам список, который для данного примера будет выглядеть, как показано на рис. 3.8.

9	11	11	3	2	2	3	6	3	6	0	9
1	2	3	4	5	6	7	8	9	10	11	12

Рис. 3.8. Список прямых предков для дерева, представленного на рис. 3.7

3.3.3 Реализация бинарных деревьев на Си

Рассмотрим вариант реализации бинарных деревьев на Си. Структуру вершины дерева можно описать следующим образом:

```
typedef struct node
{
     char *word;
     struct node * left;
     struct node * right;
} tree;
```

Функция распечатки элементов дерева в инфиксном порядке, по одному на строке, выглядит, как описано ниже:

```
void print_tree ( tree * t )
{
     if ( !t ) return;
     print_tree ( t->left );
     printf ( "%s\n", t->word );
     print_tree ( t->right );
}
```

Нетрудно заметить, что переставив соответствующим образом строку с распечаткой содержимого текущей вершины дерева, мы получим реализацию префиксного и постфиксного обходов.

Другой пример рекурсивной реализации – вычисление высоты бинарного дерева:

```
int tree_height ( tree * t )
{
     int l, r;
     if ( !t ) return -1;
     l = tree_height ( t->left );
     r = tree_height ( t->right );
     if (l > r)
          return l + 1;
     else
          return r + 1;
}
```

Описанная выше функция возвращает **-1**, если дерево пустое, т. е. если оно не имеет ни одной вершины. В противном случае результатом работы этой функции будет длина максимального пути от заданной вершины **t** до листа.

ГЛАВА 4. ПЕРЕБОРНЫЕ ЗАДАЧИ И ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Во многих прикладных задачах требуется найти оптимальное решение. Часто для его нахождения необходимо выполнить перебор всех возможных вариантов и сравнить их между собой. Поэтому важно научиться строить алгоритмы перебора различных объектов – последовательностей, перестановок, подмножеств и т. д.

Среди переборных задач можно выделить такие задачи, которые практически можно решить без полного перебора с использованием метода динамического программирования, который также рассмотрен в этой главе.

4.1 Алгоритмы с возвратом

Интересная область программирования – задачи так называемого «искусственного интеллекта», для которых решение находится не по заданным правилам вычислений, а путем проб и ошибок. Обычно процесс проб и ошибок разделяется на отдельные задачи, и они наиболее естественно выражаются в терминах рекурсии и требуют исследования конечного числа подзадач.

В общем виде весь процесс можно мыслить, как процесс поиска, строящий и обрезающий дерево подзадач. Во многих задачах такое дерево поиска растет очень быстро, его рост зависит от параметров задачи и часто бывает экспоненциальным. Иногда, используя некоторые эвристики, дерево поиска удается сократить, и свести затраты на вычисления к разумным пределам. Алгоритмы, позволяющие решать подобные задачи, называются *алгоритмами с возвратом* (*backtracking algorithms*).

Суть этих алгоритмов состоит в следующем. В них имеются точки ветвления, в которых необходимо выбрать один из вариантов поведения. В дальнейшем может оказаться, что выбранный вариант неудачен и нужно вернуться к точке ветвления и выбрать какой-то другой вариант.

Особенность этих алгоритмов состоит в том, что происходит возврат по управлению и по данным. При возврате программа «забывает» все, что она сделала в результате выбора неудачного варианта – данные восстанавливают свои предыдущие значения, отменяются все последствия выбора предыдущего варианта. После этого выбирается следующий вариант, и заново выполняются все его действия. Если в данной точке ветвления все варианты оказываются неудачными, происходит возврат на предыдущие точки ветвления.

Продемонстрируем работу такого алгоритма на хорошо известном примере – задаче о ходе коня.

4.1.1 Задача о ходе коня

Рассмотрим задачу, известную, по крайней мере, с XVIII века. Леонард Эйлер посвятил ей большую работу «Решение одного любопытного вопроса, который, кажется, не подчиняется никакому исследованию».

Постановка задачи. Дана доска размером $n \times n$. На поле с координатами (x_0, y_0) помещается конь – фигура, перемещающаяся по обычным шахматным правилам. Задача заключается в поиске последовательности ходов, при которой конь точно один раз побывает на всех полях (клетках) доски.

Если количество клеток доски нечетно, то обхода доски из некоторых клеток не существует. Отметим, что путь коня проходит по клеткам, чередующимся по цвету. Если общее число клеток доски нечетно, то первая и последняя клетки пути, пройденного конем, будут одного и того же цвета. Таким образом, обход будет существовать только тогда, когда он начнется клеткой того цвета, который имеют наибольшее число клеток. Однако приведенное правило не охватывает всех клеток, для которых обхода не существует. Так, для доски размером 3×7, помимо тех клеток, для которых выполняется приведенное правило, обход невозможен также из клетки (2, 4).

Ниже приведена схема алгоритма из работы [1], реализующего поставленную задачу. Поиск последовательности ходов считается успешным тогда, когда номер текущего хода станет равным количеству клеток на доске. Если из начальной клетки перебраны все возможные ходы, то решения не существует.

Алгоритм 4.1. Схема полного перебора для нахождения одного решения

Вход: Описание задачи.
Выход: Последовательность ходов, ведущих к решению.
Метод:

```
начало
    неудача ← истина
    Try (1)
    если неудача = ложь то
        распечатать найденную последовательность ходов
конец

процедура Try ( i : номер хода)
    инициализация выбора хода
    выполнять
        выбрать очередного хода из списка возможных
        если выбранный ход приемлем то
            начало
                запись хода
```

если ход не последний **то**
начало
Try (*i* + 1)
если *неудача* **то**
отменить предыдущий ход
конец
иначе
неудача ← *ложь*
конец
пока (*неудача*) **и** (есть другие ходы)
конец процедуры
Конец алгоритма 4.1

Для более детального описания алгоритма и его реализации на Си нужно выбрать некоторое представление для данных.

Доску можно представлять как матрицу *h*:

`int h[n][n];`

`h[x][y]` = 0 – поле с координатами (*x*, *y*) еще не посещалось;

`h[x][y]` = *i* – поле с координатами (*x*, *y*) посещалось на *i*-м ходу.

Также нужно определить параметры для начальной позиции коня, условий следующего хода и результат, если ход сделан. В первом случае достаточно задать координаты поля (*x*, *y*), откуда следует ход, и число *i*, указывающее номер хода. Условие «ход не последний» означает, что доска еще не заполнена, можно переписать как $i < n^2$.

Введем две переменные `u` и `v` для позиции возможного хода, определяемого в соответствии с правилами хода коня. Тогда условие «ход приемлем» можно представить как конъюнкцию условий, что новое поле находится в пределах доски: `0 ≤ u < n && 0 ≤ v < n`

и еще не посещалось: `h[u][v] == 0`.

Отмена хода выглядит аналогично: `h[u][v] = 0`.

Также введем локальную переменную `q` для результата, тогда получим более детализированный алгоритм.

```
int Try(int i, int x, int y)
{
      int u,v;
      int q;
      инициация выбора хода;
      do
      {     // <u,v> - координаты следующего хода;
            if( ( 0 <= u )&&( u < n )&&( 0 <= v )&&( v < n )
                  && ( h[u][v] == 0 ) )
            {
                  h[u][v]= i;
```

```
            if (i < n * n)
            {
                  q = Try(i+1,u,v);
                  if (!q1)
                        h[u][v]=0;
            }
            else
                  q1 = 1;
      }
   while(!q) && (есть другие ходы);
   return q;
}
```

При выборе хода полю с координатами (x_0, y_0) присваивается значение 1, остальным полям – 0, что свидетельствует о том, что они свободные.

Для фиксированного поля (*x, y*) количество ходов может варьироваться от двух до восьми. Ходы коня могут быть заданы, например, в виде следующего массива:

```
int D[8][2] = { {-1, -2}, {-2, -1}, {-2, 1}, { 1, -2},
                {-1, 2}, { 2, -1}, { 1, 2}, { 2, 1} };
```

Для поля **(*x*, *y*)** построим последовательность ходов:

(*x* + *D*[0][*k*], *y* + *D*[1][*k*]),

где $k = 0, 1, \dots, 7$, и отберем из них те, которые не выводят за пределы поля. На рис. 4.1 приведен фрагмент доски. Конь *K* стоит в позиции (*x, y*). Клетки с цифрами вокруг *K* – это поля, на которые конь может переместиться из **(*x*, *y*)** за один ход.

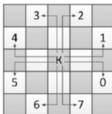


Рис. 4.1. Последовательность возможных ходов из клетки, помеченной *K*

Описанный алгоритм осуществляет перебор вариантов и находит решение, если оно имеется. Отсутствие решения приводит к полному перебору всех вариантов.

Отметим, что долгое время не было известно, справедливо ли правило Варнсдорфа (1823 г.), заключающееся в том, что на каждом ходу надо ставить коня на такое поле, из которого можно совершить наименьшее число

ходов на еще непройденные поля. Если же таких полей несколько, разрешается выбирать любое из них.

Это правило верно для доски размером от 5×5 до 76×76. Для любого другого размера доски найдены контрпримеры, построенные с помощью ЭВМ, опровергающие правило Варнсдорфа.

4.1.2 Задача о кубике

Рассмотрим следующую задачу. На гранях кубика изображены буквы. Будем считать, что буква, находящаяся на нижней стороне кубика, отпечатывается на бумаге. Перекатывая кубик с грани на грань, мы получим слово из отпечатанных букв. Повороты букв не учитываем.

Итак, нам даны описание кубика и входная строка. Можно ли получить входную строку, перекатывая кубик с грани на грань?

Для решения задачи перенумеруем грани кубика с **123456** на **124536**, а именно:

1 – нижняя грань;	**6** – верхняя;	(**1 + 6 = 7**)
3 – фронтальная;	**4** – задняя;	(**3 + 4 = 7**)
2 – боковая левая;	**5** – боковая правая	(**2 + 5 = 7**).

Тогда соседними для i-й грани будут все, кроме i-й и $(7-i)$-й. Попробуем построить слово, начиная перекатывать кубик поочередно со всех шести граней. Для хранения букв, записанных на гранях кубика, введем массив ***CB***, в ***CB*[1]** содержится буква с нижней грани, в ***CB*[2]** – с боковой левой грани и т. д. Результат работы программы будем хранить в переменной ***q***. Значение ***q*** будет равно 1, если можно получить слово, записанное в глобальной строке *w*, начиная с *n*-го символа, перекатывая кубик, лежащий на грани ***g***. Ниже приведена рекурсивная функция, реализующая это решение.

```
 int chkword(int g, int n)
 {
1     if((n > strlen(w))||(w[n]==' '))
2          return 1;        // дошли до конца строки!
3     if(CB[g] != w[n] )    // буква на грани не совпадает
           break;           // с буквой в слове
4     for (i = 1; i <= 6; i++)
      {         // начинаем поочередно со всех 6-ти граней
5          if ((i != g) && (i + g != 7))
                // если не эта грань и не противоположная
6               q = chkwrd (i, n + 1);
                // продолжим процесс с выбранной гранью
                // и следующей буквой
7          if (q)
                return 1;
      }
 }
```

В основной программе следует выполнить ввод массива ***CB*** и строки ***w*** и выполнить приведенную ниже последовательность операторов.

```
q = 0;
n = 0;
chkwrd(1,0);
```

В функции ***chkword*** в строке 1 осуществляется проверка конца заданного слова ***w***. Если смогли дойти до конца этого слова, то возвращаем значение 1. В строке 3 выполняется прерывание перекатывания кубика с грани ***g*** в том случае, если буква, записанная на этой грани, не совпадает с буквой в строке ***w***, находящейся на позиции ***n***. В строках 5–7 организована дальнейшая проверка получения оставшейся части заданного слова. При этом перекатываем кубик, поочередно начиная со всех граней, выполняя проверку на совпадение букв, записанных на соответствующей грани кубика и в слове на позиции ***n*** + 1. Результат проверки записывается в переменную ***q***.

4.1.3 Задача о стабильных браках

Заданы два непересекающихся множества A и B. Нужно найти множество пар $< a, b >$, таких что, $a \in A$, $b \in B$, удовлетворяющее некоторым условиям. Для выбора таких пар существует много различных критериев; один из них называется «правилом стабильных браков» [1].

Пусть A – множество мужчин, а B – женщин. У каждых мужчины и женщины есть различные предпочтения возможного партнера. Если среди n выбранных пар существуют мужчины и женщины, не состоящие между собой в браке, но предпочитающие друг друга, а не своих фактических супругов, то такое множество браков считается нестабильным. Если же таких пар нет, то множество считается стабильным.

Для решения этой задачи рассмотрим схему полного перебора, которая позволяет находить все решения. При этом будем иметь в виду, что на каждом ходу нужно рассмотреть n вариантов, а максимальное количество ходов в других задачах может не совпадать с количеством вариантов.

Алгоритм 4.2. Схема полного перебора для нахождения всех решений

Вход: Описание задачи.
Выход: Запись всех решений.
Метод:

начало
Try (1)
распечатать найденные решения
конец

процедура *Try* (*i* : номер хода)
инициализация выбора хода
для всех *r* **от** 0 **до** *n* – 1 **выполнить**
выбор *r*-го хода из списка возможных
если выбранный ход приемлем **то**
начало
запись хода
если ход не последний **то**
Try (*i*+1)
иначе
записать найденное решение
отменить предыдущий ход
конец
конец цикла
конец процедуры

Конец алгоритма 4.2

Алгоритм 4.2 можно адаптировать к нашей задаче, причем поиск супруги для мужчины ***m*** будет вестись в порядке списка предпочтений именно этого мужчины.

```
void Try ( int m )
{
      int r;
      for ( r = 0; r < n; r++ )
      {
            выбор r-ой претендентки для m;
            if ( подходит )
            {
                  запись брака;
                  if (m - не последний)
                        Try( m + 1 );
                  else
                        записать стабильное множество;
                  отменить брак;
            }
      }
}
```

Будем использовать две матрицы [1], задающие предпочтительных партнеров для мужчин и женщин: ***ForLady*** и ***ForMan***.

ForMan*[*m*] [*r*]** – это номер женщины, стоящей на *r*-м месте в списке для мужчины ***m.

ForLady*[*w*] [*r*]** – это номер мужчины, стоящего на ***r-м месте в списке женщины ***w***.

Результатом работы программы будет массив **x**, где **x[m]** соответствует партнерше для мужчины ***m***. Для поддержания симметрии между мужчинами и женщинами и для эффективности алгоритма будем использовать дополнительный массив ***y***: **y[*w*]** – партнер для женщины ***w***.

Условие «**подходит**» можно представить в виде конъюнкции определения стабильности брака, реализующейся функцией ***stable***, и того, что претендентка свободна, что отражается в массиве ***single***. Для уточнения стабильности нужно помнить, что оно следует из сравнения рангов, которые можно вычислить по значениям ***ForMan*** и ***ForLady***. С учетом введенных структур данных функция **Try** преобразуется к следующему виду.

```
void Try ( int m )
{
      int r, w;
      for (r = 0; r < n; r++)
      {
            w = ForMan[m][r];
            if (single[w] && stable(m, r))
            {
                  x[m] = w;
                  y[w] = m;
                  single[w] = 1;
                  if (m < n)
                        Try( m + 1 );
                  else
                        record set();
                  single[w] = 0;
            }
      }
}
```

Для стабильности системы определяется возможность брака между мужчиной ***m*** и женщиной ***w***, где ***w*** стоит в списке ***m*** на ***r***-м месте. При этом возможны следующие два случая, которые приводят к нарушению стабильности:

1) может существовать женщина ***pw***, которая для ***m*** предпочтительнее, чем ***w***, и для ***pw*** мужчина ***m*** предпочтительнее ее супруга;
2) может существовать мужчина ***pm***, который для ***w*** предпочтительнее ***m***, причем для ***pm*** женщина ***w*** предпочтительнее его супруги.

В первом случае сравниваются ранги женщин, которых ***m*** предпочитает больше, чем ***w***. Поиск осуществляется среди женщин, которые уже были выданы замуж, иначе бы выбрали ее.

```
int stable( int m, int r )
{
      int s = 1, i = 0, pw;
      while((i < r) && s)
      {
            pw = ForMan[m][i];
            i = i+1;
            if (!single[pw])
                  s = ForLady[pw][m] < ForLady[pw][y[pw]];
      }
      return s;
}
```

Во втором случае нужно проверить всех кандидатов **pm**, которые для **w** предпочтительнее «суженому». Предлагается написать эту проверку самостоятельно.

4.1.4 Метод ветвей и границ

Метод ветвей и границ можно рассматривать как вариацию полного перебора с отсечением подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Впервые этот метод был предложен А. Лендом и А. Дойгом в 1960 г. для решения общей задачи целочисленного линейного программирования. Интерес к этому методу и фактически его «второе рождение» связаны с работой Дж. Литла, К. Мурти, Д. Суини и К. Кэрола в 1963 г., посвященной задаче коммивояжера – поиска самого выгодного маршрута, проходящего через все указанные города. Начиная с этого момента, появилось большое число работ, посвященных методу ветвей и границ и различным его модификациям.

В основе метода ветвей и границ лежит идея последовательного разбиения множества допустимых решений на подмножества меньших размеров. Эти подмножества образуют дерево, называемое *деревом поиска* или *деревом ветвей и границ*. Вершинами этого дерева являются построенные подмножества.

На каждом шаге разбиения осуществляется проверка того, содержит ли данное подмножество оптимальное решение или нет. Проверка проводится посредством вычисления оценок снизу и сверху для целевой функции на данном подмножестве. Если для пары подмножеств получается такая ситуация, что *нижняя* граница для первого подмножества дерева поиска больше, чем *верхняя* граница для второго подмножества, то тогда первое подмножество можно исключить из дальнейшего рассмотрения. Если нижняя граница для вершины дерева совпадает с верхней границей, то это значение является минимумом функции и достигается на соответствующем подмножестве.

4.1.5 Использование метода ветвей и границ для решения задачи о рюкзаке

Задача о рюкзаке имеет следующую формулировку. Задается множество предметов, каждый из которых имеет определенные стоимость и вес. Требуется составить такой набор этих предметов, который имел бы суммарную стоимость, максимально возможную среди всех наборов, чей суммарный вес не превосходит заданной величины – P.

Более точно, пусть $c_i > 0$ и $w_i > 0$ – соответственно стоимость и вес i-го предмета, где $i \in \{1, 2, 3, \ldots, n\}$, а n – количество предметов. Требуется найти такой булев вектор $(x_1, x_2, \ldots, x_n)$, чтобы была максимальной сумма $\sum_{i=1}^{n} x_i c_i$, и выполнялось неравенство $\sum_{i=1}^{n} x_i w_i \le P$.

Для решения этой задачи можно применить схему из [1] перебора всех решений и выбора из них оптимального с использованием оценок снизу и сверху, представленную в алгоритме 4.3.

Алгоритм 4.3. Схема для нахождения оптимальной выборки

Вход: Описание задачи.
Выход: Оптимальное решение.
Метод:

```
    начало
        Try ( 1 )
        распечатать найденное решение
    конец

    процедура Try ( i : номер объекта)
        если включение приемлемо то              | оценка сверху
          начало
            включить i-ый объект
            если i < n то
              Try ( i + 1 )
            иначе
              проверка оптимальности
            исключить i-ый объект
          конец
        если приемлемо невключение то            | оценка снизу
          если i < n то
            Try ( i + 1 )
          иначе
            проверка оптимальности
    конец процедуры
```

Конец алгоритма 4.3

Пусть **opts** – оптимальная выборка, полученная к данному моменту, **maxv** – ее ценность, **t** – текущая выборка. Объект можно включать в выборку, если он подходит по весовым ограничениям. Критерием неприемлемости будет то, что после данного исключения общая ценность выборки будет не меньше полученного до этого момента оптимума. Если ценность меньше, то продолжение поиска не приведет к оптимальному решению.

Будем рассматривать следующие оценки: **tw** – общий вес выборки к данному моменту; **av** – общая ценность текущей выборки, которую можно еще достичь. Условие «включение приемлемо» можно сформулировать в виде выражения: $tw + w_i \le P$. Тогда проверка оптимальности будет следующей:

```
if (av > maxv)
{
   opts = t;
   maxv = av;
}
```

Условие «приемлемо невключение» проверяется с помощью выражения: $av > maxv + c_i$.

Задача о рюкзаке в общей формулировке принадлежит к классу *NP*-полных задач. Ее можно решить полным перебором всех допустимых вариантов заполнения рюкзака имеющимися предметами. Однако при больших входных данных такой переборный алгоритм практически неприемлем, поскольку имеет экспоненциальную сложность относительно размера входных данных. Если же применить идеи «метода ветвей и границ», то в большинстве случаев объем перебираемых вариантов можно сократить.

4.2 Динамическое программирование

Словосочетание *динамическое программирование* впервые было использовано в 1940-х гг. Р. Беллманом для описания процесса нахождения решения задачи, где ответ на одну задачу может быть получен только после решения задачи, «предшествующей» ей. В 1953 г. он уточнил это определение до современного. «Динамическое программирование» происходит от словосочетания «математическое программирование», которое является синонимом слова «оптимизация». Слово «программа» в данном контексте скорее означает оптимальную последовательность действий для получения решения задачи. В общем виде теория Беллмана достаточно сложна, и здесь мы ее не рассматриваем. В то же время конкретные задачи, рассмотренные ниже, демонстрируют идеи, лежащие в основе решения задач данного класса.

Идея динамического программирования похожа на подход рекурсии, но в отличие от нее, вычисления не дублируются. При динамическом

программировании задача разбивается на подзадачи, которые являются либо очевидными, либо сводятся к подзадачам, ответы которых запоминаются в таблице и используются при решении больших задач.

При такой постановке необходимо определить исходные данные задачи – параметры. Например, если мы решаем задачу нахождения суммы некоторого набора чисел, то параметрами задачи будут количество чисел и их значения. Тогда задача может быть формализована в виде некоторой функции, аргументами которой могут являться количество параметров и их значения.

Под подзадачей понимается та же постановка задачи, но с меньшим числом параметров или с тем же числом параметров, но при этом хотя бы один из параметров имеет меньшее значение.

Сведение решения задачи к решению некоторых подзадач может быть записано в виде соотношений, в которых значение функции, соответствующей исходной задаче, выражается через значения функций, соответствующих подзадачам. При этом значения аргументов у любой из функций в правой части соотношения меньше значения аргументов функции в левой части соотношения. Если аргументов несколько, то достаточно уменьшения одного из них.

Перекрывающиеся подзадачи в динамическом программировании означают подзадачи, которые используются для решения некоторого количества задач (не одной) большего размера.

Стоит отметить, что для динамического программирования характерно, что зачастую решается не заданная задача, а более общая, при этом решение исходной задачи является частным случаем решения более общей задачи.

Динамическое программирование может быть применено к задачам оптимизации. В них требуется выдать оптимальное решение, при котором значение какого-то параметра будет минимальным или максимальным, в зависимости от постановки задачи. При этом обычно требуется выписать рекуррентные соотношения, связывающие оптимальные значения параметров для подзадач, двигаясь снизу вверх. Они позволяют вычислить оптимальные решения для подзадач и на этой основе построить общее решение для поставленной задачи.

Пример

Требуется вычислить сумму s следующего ряда при $x \neq 0$:

$s = 1 + 1/x + 1/x^2 + 1/x^3 + \ldots + 1/x^n$.

Рассмотрим подзадачу: вычислить сумму заданного ряда с параметром $n = 1$. Для этого вводится переменная a, в которую будет записано значение $1/x$, а сумма ряда – в переменную s. Для решения подзадачи с параметром $n = 2$, используется значение, записанное в s: $s = s + a$, где $a = 1/x^2$. Заметим,

что новое значение a можно вычислить через старое: $a = a / x$. Затем решается подзадача с параметром $n = 3$ и т. д.

Обозначим значение a на k-м шаге ($n = k$) через $a^{(k)}$, а значение s – через $s^{(k)}$. В результате можно выписать следующие рекуррентные соотношения:

$s^{(0)} = 1$, $a^{(0)} = 1$,

$$\begin{cases} a^{(k)} = a^{(k-1)}/x \\ s^{(k)} = s^{(k-1)} + a^{(k-1)} \end{cases}$$

Как правило, при построении алгоритмов по принципу динамического программирования выполняются следующие три шага:

1) Находится такое разбиение задачи на две или более подзадач, чтобы оптимальное решение задачи содержало оптимальные решения всех подзадач, которые в нее входят.
2) Выписываются рекуррентные соотношения, и вычисляется оптимальное значение параметра для всей задачи.
3) Если необходимо получить не только значение оптимального параметра, но и найти само решение, то на шаге 2 нужно также запоминать некоторую дополнительную информацию о ходе решения – решение каждой подзадачи. Этот шаг иногда называют *обратным ходом*.

4.2.1 Задача о наибольшей общей подпоследовательности

Важным моментом при решении задачи является умение строить решение исходной задачи из решений подзадач. Одним из наиболее эффективных таких способов является использование таблиц для запоминания результатов решения подзадач.

Как было отмечено выше, задача может быть формализована посредством функции, которая зависит от одного или нескольких аргументов. Если взять таблицу, у которой количество элементов равно количеству всех возможных различных наборов аргументов функции, то каждому набору аргументов может быть поставлен в соответствие элемент таблицы. Вычислив элементы таблицы (то есть, решив подзадачи), можно найти и решение исходной задачи.

Пусть дана некоторая последовательность $X = \langle x_1, x_2, \ldots, x_m \rangle$, тогда другая последовательность $Z = \langle z_1, z_2, \ldots, z_k \rangle$ будет подпоследовательностью X, если существует такая возрастающая последовательность индексов $I = \langle i_1, i_2, \ldots, i_k \rangle$, что для всех $j = 1, 2, \ldots, k$, будет верно равенство $x_{ij} = z_j$.

Пример

$Z = \langle A, C, A, B \rangle$ – подпоследовательность последовательности $X = \langle B, A, C, B, D, A, B \rangle$ с набором индексов $\langle 2, 3, 6, 7 \rangle$.

Говорят, что Z – *общая подпоследовательность* X и Y, если она является подпоследовательностью для X и Y.

Рассмотрим задачу о нахождении наибольшей общей подпоследовательности (сокращенно НОП) [5]. Она состоит в том, чтобы найти одну из общих подпоследовательностей последовательностей X и Y наибольшей длины.

Пример

Пусть $X = <T, R, R, E, E, A>$ и $Y = <A, T, R, A, E, A, E>$, тогда последовательность $<T, R, E, A>$ – самая длинная (но не единственная) общая подпоследовательность X и Y.

Префиксом последовательности X длины i называется последовательность $X_i = <x_1, x_2, \ldots, x_i>$.

X_0 – пустая последовательность.

Теорема 4.1

Пусть $Z = <z_1, z_2, \ldots, z_k>$ – одна из наибольших общих подпоследовательностей для последовательностей:

$X = <x_1, x_2, \ldots, x_m>$ и $Y = <y_1, y_2, \ldots, y_n>$.

Тогда:

- если $x_m = y_n$, то $z_k = x_m = y_n$ и Z_{k-1} является НОП для X_{m-1} и Y_{n-1};
- если $x_m \neq y_n$ и $z_k \neq x_m$, то Z является НОП для X_{m-1} и Y;
- если $x_m \neq y_n$ и $z_k \neq y_n$, то Z является НОП для X и Y_{n-1}.

Доказательство можно найти в [5].

Из теоремы 4.1 вытекает, что самая длинная общая подпоследовательность двух последовательностей содержит в себе самую длинную общую подпоследовательность их префиксов.

Непосредственно из этой теоремы следует способ нахождения НОП двух заданных последовательностей. Для этого нужно рассмотреть два случая.

1) Если $x_n = y_m$, то построив НОП для X_{n-1} и Y_{m-1} и, присоединив к этому результату $x_n = y_m$, получим НОП для X и Y.
2) Если $x_n \neq y_m$, то нужно построить НОП для последовательностей X_{n-1} и Y и для последовательностей X и Y_{m-1}. В качестве результата следует выбрать наибольшую из двух полученных НОП.

Пусть $c[i, j]$ – длина НОП для последовательностей X_i и Y_j. Тогда длину НОП можно определить, используя следующие рекуррентные соотношения:

$$c[i,j] = \begin{cases} 0, \text{если } i = 0 \text{ или } j = 0 \\ c[i-1, j-1] + 1, \text{если } i > 0, j > 0 \text{ и } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]), \text{если } i > 0, j > 0 \text{ и } x_i \neq y_j \end{cases}$$

Для того чтобы найти саму НОП, необходимо хранить некоторую дополнительную информацию. Ниже представлен алгоритм 4.4, который

вычисляет длину НОП. В нем поддерживается двумерная таблица b[0 .. m , 1 .. n], с использованием которой упрощается процесс построения НОП. Элемент $b[i, j]$ указывает на тот элемент таблицы, который соответствует подзадаче, использованной для нахождения $c[i, j]$. В процессе работы этого алгоритма создаются таблицы b и c. В $c[m, n]$ записана длина НОП. Время работы алгоритма равно $O(m \cdot n)$.

Алгоритм 4.4. Вычисление длины НОП

Вход: $X = <x_1, x_2, \dots , x_m>$, $Y = <y_1, y_2, \dots , y_n>$ – последовательности,
m, n – длины последовательностей X и Y, соответственно.
Выход: длина НОП последовательностей X и Y.
Метод:

```
для всех i от 0 до m выполнить
    c [i, 0] ← 0                          | заполнение нулевого столбца
конец цикла
для всех j от 0 до n выполнить
    c [0, j] ← 0                          | заполнение нулевой строки
конец цикла
для всех i от 0 до m выполнить
    для всех j от 0 до n выполнить
        если x_i = y_j то
            начало   | использована длина НОП префиксов X_{i–1} и Y_{j–1}
                c [i, j] ← c [i –1, j –1] +1
                b [i, j] ← '↖'            |
            конец
        иначе
            если c [i –1, j ] ≥ c [i , j –1] то
                начало | использована длина НОП префиксов X_{i–1} и Y_j
                    c [i, j] ← c [i –1, j]
                    b [i, j] ← '↑'
                конец
            иначе
                начало | использована длина НОП префиксов X_i и Y_{j–1}
                    c [i, j] ← c [i, j –1]
                    b [i, j] ← '← '
                конец
    конец цикла
конец цикла
выдать c[m, n]
```

Конец алгоритма 4.4

Пример

Для последовательностей $X = <T, R, R, E, E, A>$ и $Y =<A, T, R, A, E, A, E>$ на рис. 4.2 совмещены две таблицы *c* и *b*. В элементе *c*[7, 6] находится значение 4. Это и есть максимальная длина НОП для *X* и *Y*.

		0	1	2	3	4	5	6
		y_j	***T***	*R*	***R***	*E*	***E***	***A***
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑0	↑0	↑0	↑0	↑0	↑1
2	***T***	0	↖1	←1	←1	←1	←1	↑1
3	***R***	0	↑1	↖2	↖2	←2	←2	←2
4	*A*	0	↑1	↑2	↑2	↑2	↑2	↖3
5	***E***	0	↑1	↑2	↑2	↖3	↖3	↑3
6	***A***	0	↑1	↑2	↑2	↑3	↑3	↖4
7	*E*	0	↑1	↑2	↑3	↖3	↖4	↑4

Рис. 4.2. Таблицы *c* и *b* для последовательностей
$X = <T, R, R, E, E, A>$ и $Y = <A, T, R, A, E, A, E>$

Обратный ход

Чтобы получить элементы НОП следует пройти по стрелкам таблицы *b*, ведущим из правого нижнего угла. Если значением *b*[*i*, *j*] является элемент '↖', то *x*[*i*] = *y*[*j*] принадлежит НОП. Элементы НОП, восстановленные таким способом, идут в обратном порядке.

Ниже приведена процедура *печать_НОП*, которая выводит элементы НОП в прямом порядке. В программе вызов этой процедуры выполняется с параметрами: *печать_НОП* (*b*, *X*, *m*, *n*).

```
процедура печать_НОП (b, X, i, j)
    если i = 0 или j = 0 то
        выход
    если b[i, j] = '↖' то
        начало
            печать_НОП (b, X, i – 1, j – 1)
            печать (xi);
        конец
    иначе
        если b[i, j] = '↑' то
            печать_НОП (b, X, i – 1, j)
        иначе
            печать_НОП (b, X, i, j – 1)
конец процедуры
```

Для таблицы b (рис. 4.2) процедура выводит последовательность **<*T*, *R*, *E*, *A*>**. Время работы процедуры равно $O(n+m)$.

Заметим, что при решении этой задачи возможно обойтись без таблицы b. Каждый элемент $c[i, j]$ определяется по значениям трех элементов $c[i-1, j-1]$, $c[i-1, j]$ и $c[i, j-1]$. Можно за константное время определить, какое из этих значений было использовано, не прибегая к таблице b.

4.2.2 Задача о преобразовании строк

Рассмотрим следующую задачу. Пусть даны две строки S_1 и S_2. Необходимо за минимальное число *допустимых* операций преобразовать строку S_1 в строку S_2. Допустимой операцией являются следующие операции удаления символа из строки и вставки символа в строку:

DEL(S, i) – удалить i-ый элемент строки S;

INS(S, i, c) – вставить символ c после i-го элемента строки S.

Обозначим через $S[i .. j]$ подстроку строки S от i-го символа до j-го. $S_1[0 .. 0]$ и $S_2[0 .. 0]$ – пустые строки.

Пусть $M[i, j]$ – минимальное количество операций, которые требуются, чтобы преобразовать начальные i символов строки S_1 (подстроку $S_1[0 .. i]$), в начальные j символов строки S_2 (подстроку $S_2[0 .. j]$).

Заметим, что для преобразования пустой строки в строку длины j требуется j операций вставки, т. е. $M[0, j] = j$. Аналогично, для преобразования строки длины i в пустую строку требуется i операций удаления, т. е. $M[i, 0] = i$.

Пусть решена подзадача для строк с параметрами $i-1$ и $j-1$. Это означает, что из строки $S_1[0 .. i-1]$ построена строка $S_2[0 .. j-1]$ за минимальное число допустимых операций $M[i-1, j-1]$.

Рассмотрим два случая:

1) Пусть i-ый символ строки S_1 равен j-му символу строки S_2, т. е. $S_1[i] = S_2[j]$. Тогда для получения строки $S_2[0 .. j]$ из строки $S_1[0 .. i]$ не требуется никаких дополнительных операций, достаточно тех, что были выполнены для получения строки $S_1[0 .. i-1]$ из $S_2[0 .. j-1]$. Следовательно, в этом случае $M[i, j] = M[i-1, j-1]$.

2) Пусть $S_1[i] \neq S_2[j]$. Возможны два способа получения строки $S_2[0 .. j]$.

Пусть из строки $S_1[0 .. i-1]$ построена строка $S_2[0 .. j]$ за минимальное количество операций $M[i-1, j]$. Тогда для получения строки $S_2[0 .. j]$ из строки $S_1[0 .. i]$ требуется удалить i-ый символ из строки S_1.

Теперь, пусть из строки $S_1[0 .. i]$ построена строка $S_2[0 .. j-1]$ за минимальное количество операций $M[i, j-1]$. Тогда для получения строки $S_2[0 .. j]$ из строки $S_1[0 .. i]$ потребуется одна операция вставки i-го символа строки S_1 после символа $S_2[j-1]$. Понятно, что из этих возможностей необходимо выбрать лучшую.

Таким образом, получаем следующие рекуррентные соотношения:

$M[0, j] = j$;

$M[i, 0] = i$;

$$M[i,j] = \begin{cases} \min\begin{pmatrix} M[i-1,j-1] \\ M[i-1,j]+1 \\ M[i,j-1]+1 \end{pmatrix}, \text{если } S_1[i] = S_2[j] \\ \min\begin{pmatrix} M[i-1,j] \\ M[i,j-1] \end{pmatrix} + 1, \text{если } S_1[i] \neq S_2[j] \end{cases}$$

Решением задачи будет значение $M[m, n]$, где m – длина строки S_1, а n – длина строки S_2.

Обратный ход

По построенной таблице можно также определить, какие операции и в каком порядке применяются к строке S_1. Обратный ход будет состоять в следующем. Нужно встать на последний символ строки S_1 и параллельно начать движение по таблице от правого верхнего угла, т. е. из ячейки $M[m, n]$.

Находясь в ячейке $M[i, j]$, будем рассматривать три рядом расположенные ячейки: $M[i-1, j-1]$, $M[i, j-1]$, $M[i-1, j]$. Всегда будем смещаться в ту из них, в которой значение наименьшее. Если такой ячейкой будет $M[i-1, j-1]$, то будем перемещаться по строке S_1 на один символ влево, т.е. сделаем текущим в строке символ, находящийся в $i-1$ позиции. При движении влево по таблице в ячейку $M[i-1, j]$ будем удалять i-й символ в строке S_1, перемещаясь на один символ влево в ней. При движении вниз по таблице в ячейку $M[i, j-1]$ будем вставлять после i-го символа в строке S_1 символ $S_2[j]$.

Движение по таблице и соответствующие ему операции представлены в таблице 4.1.

Таблица 4.1

Соответствие операций движению по таблице

Движение		*Операция*	
↓	вниз по i-му столбцу из j-ой строки в j–1-ю	INS(S_1, i, $S_2[j]$)	вставка после i-й позиции в S_1 символа $S_2[j]$
←	движение влево по j-й строке из i-го столбца в (i–1)-й	DEL(S_1, i)	удаление i-го символа в S_1 и передвижение на (i –1)-ю позицию
↙	движение по диагонали влево вниз		перемещение текущей позиции в S_1 на один символ влево

Отметим, что решений в данной задаче для произвольных входных данных может быть несколько.

Пример

Рассмотрим строки S_1 = **"abc"**, S_2 = **"aabddc"**. Построим таблицу M, нумерация строк и столбцов которой начинается с нуля. В данном случае таблица состоит из семи строк и четырех столбцов (рис. 4.3).

	0	1	2	3
c	6	5	4	3
d	5	4	3	4
d	4	3	2	3
b	3	2	1	2
a	2	1	2	3
a	1	0	1	2
	0	1	2	3
		a	**b**	**c**

Рис. 4.3. Таблица M для примера

В позиции, $M[i, j]$ будет находиться число, соответствующее минимальному количеству операций, преобразовывающих начальные i символов строки S_1, в начальные j символов строки S_2. Например, $M[1, 3] = 2$, означает, что из строки *"a"* можно получить строку *"aab"*, используя две допустимых операции. Решением задачи является число, полученное в правом верхнем углу таблицы: для нашего примера оно равно 3. Таким образом, за три допустимых операции можно преобразовать строку S_1 в S_2.

Обратный ход

Опишем последовательность действий для нашего примера. Изначально текущим в строке S_1 является последний символ – символ *'c'*. В таблице эта ситуация соответствует ячейке $M [3, 6]$. Так как $M[2, 5]$ имеет минимальное значение из трех значений, окружающих текущую ячейку, то осуществляем переход по диагонали в ячейку $M[2, 5]$ и текущим в S_1 становится предпоследний символ – *'b'*.

Далее, так как $M[2, 4] = \min(M[2, 4], M[1, 4], M[1, 5])$, передвигаемся в ячейку $M[2, 4]$. При этом вставим после текущего символа *'b'* символ $S_2[5]$ = *'d'* (j = 5). Продолжая этот процесс, вставим символ S_2 [4] = *'d'*, затем в строке S_1 сделаем текущим символ *'a'*, после чего вставим в строку S_1 символ *'a'*. Процесс продолжается до тех пор, пока не достигнем ячейки $M[0, 0]$. Для нашего примера последовательность операций будет следующая: INS(S_1, 2, *'d'*), INS(S_1, 2, *'d'*), INS(S_1, 1, *'a'*).

Задача «Телефонный номер»

Эта задача была предложена жюри Всероссийской олимпиады школьников на областной тур. Ниже приведена ее формулировка.

Если вы обратили внимание, то клавиатура многих телефонов выглядит так, как показано на рис. 4.4. Использование изображенных на клавишах букв позволяет представить номер телефона в виде легко запоминающихся слов, что бывают часто более удобным, чем традиционная запись телефона в виде последовательности цифр. Многие фирмы пользуются этим и стараются подобрать себе номер телефона так, чтобы он содержал как можно больше букв из имени фирмы.

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MN
7 PRS	8 TUV	9 WXY
	0 OQZ	

Рис. 4.4. Вид клавиатуры телефона

Требуется написать программу, которая преобразует исходный цифровой номер телефона в соответствующую последовательность букв и цифр, содержащую как можно больше символов из названия фирмы. При этом буквы из названия фирмы должны быть указаны в полученном номере в той же последовательности, в которой они встречаются в названии фирмы. Например, если фирма называется ***IBM***, а исходный номер телефона – 246, то замена его на ***BIM*** не допустима, тогда как замена его на ***2IM*** или ***B4M*** является правильной.

Упражнение

Рекомендуем решить задачу о телефонном номере самостоятельно. При ее решении можно использовать описанный выше алгоритм.

Например, для приведенного в условии примера, задайте строки S_1 и S_2: S_1= "IBM", S_2= "246". Обратите внимание, что операция INS будет вставлять цифры. При этом если в S_1 встречаются буквы, которые соответствуют цифрам номера телефона в нужном порядке, то они останутся без изменения.

Другой способ решения этой задачи – использование алгоритма поиска наибольшей общей подпоследовательности.

4.2.3 Задача о рюкзаке

Вернемся к формулировке задачи о рюкзаке из п. 4.1.5. Напомним, что задача состоит в том, чтобы определить наиболее ценную выборку из n предметов, подлежащих упаковке в рюкзак, имеющий ограничение по весу, равное P килограмм. При этом i-й предмет характеризуется стоимостью c_i и весом w_i, где $i \in \{1, 2, 3, \ldots, n\}$.

Если перебирать все подмножества данного набора из n предметов, то получится решение сложности не менее чем $O(2^n)$. В настоящее время неизвестен алгоритм решения этой задачи, сложность которого является полиномиальной. Мы рассмотрим алгоритм решения данной задачи для случая, когда все входные данные – целые числа. Его быстродействие оценивается как $O(n \cdot P)$.

Пусть $T[i, j]$ – максимальная стоимость выборки из первых i предметов, вес которой не превышает j килограммов. Здесь i и j – целые числа и $0 \leq i \leq n$, $0 \leq j \leq P$. Тогда максимальная стоимость искомого набора равна $T[n, P]$.

Определим начальные значения таблицы T следующим образом.

$T[0, j] = 0$, при $j \geq 0$ (нет предметов, максимальная стоимость равна 0),

$T[i, 0] = 0$, при $i \geq 0$ (можно брать любые из первых i предметов, но ограничение по весу равно 0).

Для решения подзадачи, соответствующей элементу таблицы $T[i, j]$, рассмотрим два случая.

1) Если i-ый предмет не упаковывается в рюкзак, то решение задачи с i предметами сводится к решению задачи с $i - 1$ предметом, т. е. $T[i, j] = T[i - 1, j]$.

2) Если же i-ый предмет упаковывается в рюкзак, то вес оставшихся предметов уменьшается на величину w_i, а при добавлении i-го предмета стоимость выборки увеличивается на c_i.

Следовательно, $T[i, j] = T[i - 1, j - w_i] + c_i$. При этом нужно учитывать, что такая ситуация возможна только тогда, когда вес i-го предмета не больше значения j.

Теперь для оптимального решения из двух возможных вариантов упаковки рюкзака нужно выбрать наилучший. Рекуррентные соотношения при $i \geq 1$ и $j \geq 1$ имеют следующий вид:

$$T[i,j] = \begin{cases} T[i-1,j], \text{при } j < w_i \\ \max(T[i-1,j], T[i-1,j-w_i] + c_i), \text{при } j \geq w_i \end{cases}$$

Ниже приведен алгоритм 4.5 вычисления выборки максимальной стоимости.

Алгоритм 4.5. Вычисление выборки максимальной стоимости

Вход: P – максимальный вес,
n – количество предметов,
$C = < c_1, c_2, \dots, c_n >$ – набор стоимостей предметов,
$W = < w_1, w_2, \dots, w_n >$ – набор весов предметов.

Выход: максимальная стоимость выборки при заданном ограничении на вес.

Метод:

```
для всех j от 0 до P выполнить
    T[0, j] ← 0    | максимальная стоимость из 0 предметов равна 0
конец цикла
для всех i от 0 до n выполнить
    T[i, 0] ← 0                     | ограничение по весу равно 0
конец цикла
для всех i от 0 до n выполнить
    для всех j от 0 до P выполнить
        если wi ≤ j то   | вес текущего i-го предмета не превышает j
                         | пытаемся добавить i-ый предмет в рюкзак
            T[i, j] ← max (T[i – 1, j], T[i –1, j – wi] + ci)
        иначе
            T[i, j] ← T[i –1, j]    | не кладем i-ый предмет в рюкзак
    конец цикла
конец цикла
выдать T[n, P]
```

Конец алгоритма 4.5

Обратный ход

Как определить тот набор предметов, которые подлежат упаковке в рюкзак? Сравним значение таблицы $T[n, P]$ со значением $T[n - 1, P]$.

Если $T[n - 1, P] \neq T[n - 1, P]$, то предмет с номером n обязательно упаковывается в рюкзак, после чего переходим к сравнению элементов таблицы $T[n - 1, P - w_n]$ и $T[n - 2, P - w_n]$ и т. д.

Если же $T[n - 1, P] = T[n - 1, P]$, то n-ый предмет можно не упаковывать в рюкзак, так как максимальная стоимость набрана без него. В этом случае следует перейти к рассмотрению элементов $T[n - 1, P]$ и $T[n - 2, P]$.

Ниже описана процедура *Печать_выборки*(i, j), которая распечатывает номера предметов, вошедших в набор, подлежащий упаковке в рюкзак с общим весом не более j килограммов, составленный из первых i предметов. В программе вызов этой процедуры выполняется с параметрами:

Печать_ выборки (n, P).

```
процедура Печать_ выборки ( i, j )
    если T[i, j] = 0 то
        выход
    иначе
        если (T[i – 1, j] = T[i, j]) то
                                | можно составить рюкзак без i-го предмета
            Печать_ выборки ( i – 1, j )
        иначе                   | предмет с номером i упакован в рюкзак
            начало
                Печать_ выборки ( i – 1, j – wi)
                печать( i )          | печать i-го предмета
            конец
конец процедуры
```

Данная процедура выдает один из возможных наборов. Предлагается самостоятельно определить, как ее изменить, чтобы выдавались все возможные наборы максимальной стоимости.

Пример

Пусть заданы ограничение общего веса $P = 15$ кг и следующие значения стоимости и веса для пяти предметов:

$c_1 = 5,\ w_1 = 4;$

$c_2 = 7,\ w_2 = 5;$

$c_3 = 3,\ w_3 = 3;$

$c_4 = 9,\ w_4 = 6;$

$c_5 = 8,\ w_5 = 6.$

Решение нашего примера определяется как $T[5, 15] = 21$. В данном случае суммарный вес предметов, подлежащих упаковке в рюкзак, совпадает с P, в общем же случае он не должен превосходить величину P.

Таблица 4.2

Значения таблицы *T*

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5
2	0	0	0	0	5	7	7	7	7	**12**	12	12	12	12	12	12
3	0	0	0	4	5	7	7	9	11	12	12	12	15	15	15	15
4	0	0	0	4	5	7	7	9	10	12	14	16	16	16	19	**21**
5	0	0	0	4	5	7	8	9	11	12	14	16	16	17	19	**21**

Обратный ход

В нашем примере $T[5, 15] = T[4, 15]$, поэтому пятый предмет в рюкзак не упаковывается. Переходим к сравнению элементов таблицы $T[4, 15]$ и

$T[3, 15]$. Их значения не равны, следовательно, четвертый предмет должен быть включен в искомый набор, а ограничение на вес становится равным $15 - w_5 = 15 - 6 = 9$.

Далее сравниваются элементы таблицы $T[3, 9]$ и $T[2, 9]$, они равны, следовательно, третий предмет в рюкзак не упаковывается. $T[2, 9]$ и $T[1, 9]$ не совпадают, следовательно, второй предмет должен быть взят в рюкзак, а ограничение на вес становится равным $9 - w_2 = 9 - 5 = 4$. И последнее, сравниваются элементы таблицы $T[1, 4]$ и $T[0, 4]$, они не равны, поэтому второй предмет включатся в искомый набор, при этом ограничение по весу становится равным 0. Итак, для нашего примера в рюкзак упакуются предметы с номерами 1, 2 и 4.

Заметим, что рассуждения были приведены для случая, когда все предметы различны. Предлагаем самостоятельно рассмотреть, какие изменения будут внесены в таблицу T в случае, если в наборе могут встречаться одинаковые предметы.

4.2.4 Задача о горнолыжных соревнованиях [1]

Горнолыжник, готовясь к соревнованиям, нарисовал на бумаге схему горнолыжной трассы для выбора оптимального маршрута спуска. Расположенные на трассе ворота представлены на схеме горизонтальными отрезками. Ни одна пара ворот не имеет общих точек.

Маршрут должен представлять собой ломаную, начинающуюся в точке старта на вершине горы и заканчивающуюся в точке финиша у ее подножия. Маршрут выбирается таким образом, что y-координата каждой следующей вершины ломаной оказывается строго меньше y-координаты предыдущей вершины.

За каждые ворота, через которые не проходит маршрут, лыжнику начисляются штрафные очки. Общий штраф за спуск по маршруту вычисляется как сумма длины маршрута и штрафных очков за непройденные ворота.

Требуется написать программу, которая определяет, какой минимальный общий штраф горнолыжник может получить при прохождении трассы.

Решение

Рассмотрим некоторую траекторию лыжника, удовлетворяющую условиям задачи. Пусть S – точка старта, а F – точка финиша с координатами (S_x, S_y) и (F_x, F_y) соответственно. Назовем *ключевыми точками* концы ворот и точки S и F. Заметим, что число ключевых точек равно $2 \cdot N+2$, где N – количество ворот на трассе. Можно показать, что всегда существует ломаная линия с минимальным общим штрафом, проходящая только через ключевые точки.

---

[1] Задача предлагалась участникам на Всероссийской олимпиаде школьников по информатике в 2004 г., автор задачи – Т. Г. Чурина

Эту задачу можно решать с помощью метода динамического программирования. Решение подзадачи означает определение общего штрафа за спуск по ломаной линии, заканчивающейся в ключевой точке, y-координата которой строго больше y-координаты точки финиша.

Сначала ключевые точки сортируются по убыванию высоты, и для каждой из них вычисляется общий штраф за спуск по маршруту, начинающемуся в точке старта S и оканчивающемуся в этой точке. Для точки S общий штраф равен сумме штрафов всех ворот.

Общий штраф для точек, которые на плане лежат строго ниже точки старта определяется следующим образом. Рассмотрим некоторый маршрут P, начинающийся в точке S и заканчивающийся в ключевой точке A. Общий штраф маршрута P определяется как сумма длины этого маршрута и штрафных очков тех ворот, которые не пересекают данный маршрут или пересекают его только в точке A. Для определения искомой величины посчитаем для всех ключевых точек A вещественную величину $T[A]$ – общий штраф ломаной линии, проходящей от точки старта S до точки A с узлами только в ключевых точках.

Тогда ответом задачи будет значение $T[F]$.

Итак, $T[S] = \sum_{i=1}^{N} c_i$ – сумма штрафов всех ворот. Пусть A – некоторая ключевая точка, отличная от S, и пусть значение $T[B]$ посчитано для всех ключевых точек B с ординатами большими, чем у A. В оптимальном маршруте лыжника от S до A существует некоторый узел B, непосредственно предшествующий A, при этом ордината B строго больше ординаты A. Если $w(BA)$ – общий штраф ломаной, начинающейся в точке B и заканчивающейся в точке A напрямую, то общий штраф ломаной линии, идущей из S в точку B, а затем напрямую в точку A, равен $T[B]+w(BA)$.

Таким образом, формула для подсчета величины $T[A]$ будет следующей.

$T[A] = min\{T[B]+w(BA)$, где B – ключевая точка с ординатой большей, чем у точки $A\}$

Ниже представлен алгоритм вычисления величины $T[F]$.

Алгоритм 4.6. Решение задачи о горнолыжных соревнованиях

Вход: N – количество ворот на трассе, S_x, S_y, F_x, F_y – координаты точек старта и финиша соответственно, a_i, b_i, y_i, c_i – x-координаты левого и правого концов ворот, y-координата ворот и штраф за непрохождение данных ворот ($a_i < b_i$, $F_y < y_i < S_y$, c_i – целое число, $0 \le c_i \le 10000$). Все координаты – целые числа, не превосходящие по модулю 10000.

Выход: минимальный общий штраф за прохождение трассы.

Метод:

начало

1 $T[S] \leftarrow$ сумма штрафов всех ворот

2 **для всех** ключевых точек $A \neq S$ в порядке убывания ординаты **выполнить**

3 $T[A] \leftarrow \infty$

конец цикла

4 **для всех** ключевых точек B с ординатой большей, чем у A **выполнить**

5 $time \leftarrow T[B] + w(BA)$

6 **если** $time < T[A]$ **то**

7 $T[A] \leftarrow time$

конец цикла

8 **выдать** $T[F]$

конец

Конец алгоритма 4.6

Обозначим величиной $c(BA)$ сумму штрафов ворот, пересекающихся с отрезком BA не в точке A, а $|BA|$ – длину ломаной линии, начинающейся в точке B и заканчивающейся в точке A. Значение $w(BA)$ в строке 5 вычисляется за время $O(N)$. Это следует из того, что $w(BA) = |BA| - c(BA)$, и для каждых ворот определяется, пересекают ли они отрезок BA не в точке A, после чего суммируются штрафы этих ворот. Так как строка 5 выполняется $O(N^2)$ раз, то время работы алгоритма есть $O(N^3)$.

ГЛАВА 5. СТРУКТУРЫ ДАННЫХ ПОИСКА

5.1 Деревья двоичного поиска

Деревом двоичного поиска для множества S называется помеченное двоичное дерево, каждая вершина v которого помечена элементом $l(v) \in S$ так, что

- $l(u) < l(v)$ для каждой вершины u из левого поддерева вершины v,
- $l(w) > l(v)$ для каждой вершины w из правого поддерева вершины v,
- для любого элемента $a \in S$ существует единственная вершина v, такая что $l(v) = a$.

Пример

Пусть $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Тогда дерево двоичного поиска для этого множества может выглядеть, например, так, как показано на рис. 5.1.

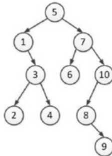


Рис. 5.1. Дерево двоичного поиска для примера

В алгоритме 5.1 приведен вариант обхода дерева двоичного поиска с целью определения, есть ли искомый элемент в том множестве, для которого это дерево построено.

Алгоритм 5.1. Просмотр дерева двоичного поиска

Вход: Дерево T двоичного поиска для множества S, r – корень дерева T, элемент a.

Выход: *истина*, если $a \in S$, *ложь* – в противном случае.

Метод:

начало

  если $T = \varnothing$ **то выдать** *ложь*

  иначе выдать *Поиск* (a, r)

конец

функция *Поиск* (a, v): *логический*

| a – искомый элемент, v – вершина дерева

если $a = l(v)$ **то**

выдать *истина*

иначе

если $a < l(v)$ **то**

если v имеет левого сына w **то**

выдать *Поиск* (a, w)

иначе

выдать *ложь*

иначе

если v имеет правого сына w **то**

выдать *Поиск* (a, w)

иначе

выдать *ложь*

конец функции

Конец алгоритма 5.1

Заметим, что данный алгоритм не трудно преобразовать в алгоритм построения дерева двоичного поиска.

5.2 Сбалансированные деревья

Максимальное число сравнений, необходимых для построения дерева двоичного поиска из n элементов, оценивается как $O(n^2)$. Эта оценка справедлива для вырожденных деревьев. Поэтому необходимо так строить деревья поиска, чтобы выполнялась оценка, приведенная в теореме 5.1.

Теорема 5.1

Среднее число сравнений, необходимых для вставки n случайных элементов в дерево двоичного поиска, пустое в начале, равно $O(n\ log_2 n)$ для $n \geq 1$.

Без доказательства

Дерево двоичного поиска называется *сбалансированным* тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу.

Такие деревья имеют второе название – *АВЛ-деревья*, по имени авторов, которые предложили их к изучению (1964 г. – Г. М. Адельсон-Вельский и Е. М. Ландис). Пример сбалансированного дерева показан на рис. 5.2. В каждой из вершин этого дерева указана метка – ее высота.

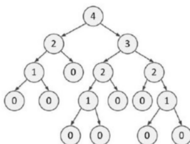


Рис. 5.2. Пример сбалансированного дерева

При построении сбалансированного дерева двоичного поиска необходимо, чтобы дерево оставалось сбалансированным на каждом шаге. Поэтому важно правильно выполнять операции вставки элемента в дерево и удаления из него. Рассмотрим на примере из [1], как они выполняются.

На рис. 5.3. представлено дерево, в котором r – корень, L – левое поддерево, R – правое поддерево. Пусть нам требуется включить новый элемент в поддерево L, и пусть включение нового элемента приведет к увеличению высоты поддерева L на 1.

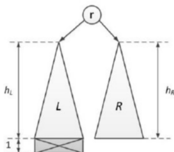


Рис. 5.3. Вставка в сбалансированное дерево нового элемента приводит к увеличению высоты левого поддерева на 1

Возможны три случая:

1) $h_L = h_R$, в этом случае дерево остается сбалансированным;
2) $h_L < h_R$, в этом случае дерево также остается сбалансированным;
3) $h_L > h_R$, в этом случае нарушен принцип сбалансированности, поэтому дерево нужно перестраивать.

Возможны два варианта вставки элемента в поддерево L.

1) При вставке элемента в левое поддерево, обозначенное на рис. 5.4 через 1, преобразование дерева состоит в том, что корнем дерева становится вершина A, левым поддеревом становится поддерево, обозначенное 1,

а правым поддеревом становится поддерево с корнем *B*, его правым и левым поддеревьями становятся поддеревья 2 и 3, соответственно. Авторы этого учебного пособия предлагают самостоятельно проверить выполнение свойств сбалансированности перестроенного дерева.

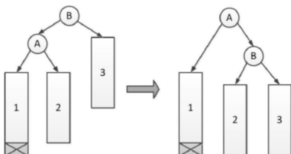


Рис. 5.4. Преобразование сбалансированного дерева при вставке нового элемента в левое поддерево левого поддерева

2) При вставке элемента в поддерево 2 или 3, выполняется следующее преобразование дерева. Корнем дерева становится вершина *B*, левым поддеревом становится поддерево с корнем в вершине *A*, а его поддеревьями становятся поддеревья, обозначенные на рис. 5.5 через 1 и 2. Правым поддеревом становится поддерево с корнем *C*, его правым и левым поддеревьями становятся поддеревья 3 и 4, соответственно. Рекомендуется проверить самостоятельно выполнение свойств сбалансированности преобразованного дерева.

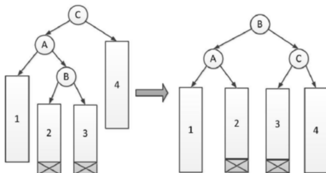


Рис. 5.5. Преобразование сбалансированного дерева при вставке нового элемента в правое

5.3 B-деревья

B-деревья – это сбалансированные деревья, обеспечивающие эффективное хранение информации на дисках и других устройствах с прямым доступом [5]. Использование B-деревьев впервые было предложено Р. Бэйером (*R. Bayer*) и Е. МакКрейтом (*E. McCreight*) в 1970 г.

B-дерево – корневое дерево, устроенное следующим образом:

1. Каждая вершина x содержит поля, в которых хранятся:
   a) $n[x]$ – количество ключей, хранящихся в данной вершине;
   b) сами ключи $key_0[x] \le key_1[x] \le ... \le key_{n[x]-1}[x]$ в неубывающем порядке;
   c) булевское значение $leaf$[x], истинное, если вершина x – лист.
2. Если x – внутренняя вершина, то она также содержит $(n[x] + 1)$ указателей: $C_0[x]$, $C_1[x]$, … , $C_{n[x]}[x]$ на ее сыновей.
   Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях:
   $$k_0 \le key_0[x] \le k_1 \le key_1[x] \le ... \le key_{n[x]-1}[x] \le k_{n[x]},$$
   где k_i – множество ключей, хранящихся в поддереве с корнем $C_i[x]$.
3. Все листья находятся на одной и той же глубине, равной высоте дерева.
4. Число ключей, хранящихся в одной вершине, ограничено сверху и снизу единым для B-дерева числом $t \ge 2$, которое называется *минимальной степенью B-дерева*:
   a) Каждая вершина, кроме корня, содержит, по меньшей мере, t – 1 ключей. Таким образом, внутренняя вершина (кроме корня) имеет t сыновей. Если дерево не пусто, то в корне должен храниться хотя бы один ключ.
   b) В каждой вершине хранится не более $2 \cdot t - 1$ ключей, внутренняя вершина имеет не более $2 \cdot t$ сыновей.

Вершину, хранящую $2 \cdot t - 1$ ключей, называют *полной*. Если $t = 2$, то у каждой вершины 2, 3 или 4 сына. Такое дерево называется 2-3-4 *деревом*. На рис. 5.6 приведен пример из [5] 2-3-4 дерева, в котором выделены полные вершины.

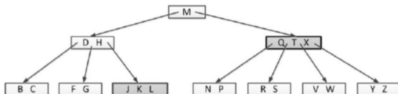


Рис. 5.6. 2-3-4 дерево

Для эффективной работы t надо брать достаточно большим. Например, на рис. 5.7 B-дерево из [5] высоты 2 содержит более миллиарда ключей. В нем каждая вершина содержит по 1000 ключей и более миллиона листьев на глубине 2.

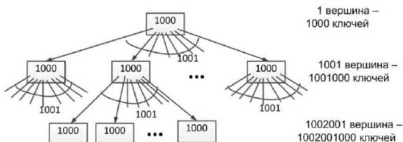


Рис. 5.7. B-дерево, каждая вершина которого содержит 1000 ключей

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, то есть каждой вершине дерева соответствует блок внешней памяти (страница). У таких деревьев, как правило, только корень находится в оперативной памяти, остальное дерево – на диске. Диск разбит на сектора. Обычно записывают или считывают сектор целиком. Время доступа, чтобы подвести головку к нужному месту на диске, может быть достаточно большим. Как только головка диска установлена, запись или чтение происходит довольно быстро. Часто получатся, что обработка прочитанного занимает меньше времени, чем поиск нужного сектора. Поэтому важно количество обращений к диску [5].

Итак, каждая вершина x в B-дереве хранит $n[x]$ ключей и имеет $n[x] + 1$ сыновей. Ключи служат границами, разделяющими всех ее потомков на $n[x] + 1$ групп. При поиске в B-дереве мы сравниваем искомый ключ с $n[x]$ ключами из x и по результатам сравнения выбираем один из $n[x] + 1$ путей.

Ниже показано описание структуры дерева, создание корня дерева с ключом 'M' и одного сына при предположении, что корень и вершины хранятся в оперативной памяти. Ссылки на сыновей вершины x обозначены через $C_i[x]$.

```
typedef struct tree        // описание структуры дерева
{
     int n;                // количество ключей
     int *key;             // key[0] ≤ key[1] ≤ … ≤ key[n-1]
     struct tree **child;  // указатели на сыновей
} B_tree;
```

```
B_tree *B =(B_tree *) malloc (sizeof(B_tree));
B->key = (int *) malloc (sizeof(int));
B->n = 1;                              // один
B->key[0] = 'M';                       // ключ равен 'M'
B->child = NULL;                       // сыновей нет
```

Далее приведено описание создания двух вершин C_0 и C_1, являющихся сыновьями корня.

```
B->child = (B_tree **) malloc ( sizeof(B_tree *) * 2 );
B->child[0] = (B_tree *) malloc ( sizeof(B_tree) );
B->child[1] = (B_tree *) malloc ( sizeof(B_tree) );
x = B->child[0];
x->n = 2;
x->key = (int *) malloc ( 2 * sizeof(int) );
x->key[0] = 'D';
x->key[1] = 'H';
x->child = NULL;
```

Аналогичные действия требуется выполнить для создания вершины C_1 с ключами Q, T, X. Результат приведенных действий показан на рис. 5.8.

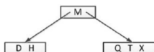


Рис. 5.8. В-дерево с двумя сыновьями

Можно выполнить реализацию В-дерева с использованием файлов, в которой каждый сын вершины дерева хранится в отдельном файле.

Также как в [5] будем считать, что имеются операции:

– `Disk_READ(x)` – чтение с диска вершины x;
– `Disk_Write(x)` – запись на диск вершины x.

Будем учитывать только количество обращений к диску.

5.3.1 Алгоритм поиска

Теорема 5.2

Для любого В-дерева высоты h и минимальной степени $t \geq 2$, хранящего $n \geq 1$ ключей, выполнено неравенство:

$$h \leq log_t \frac{n+1}{2}.$$

Без доказательства

Высота B-дерева с n-вершинами есть $O(log\ n)$. По сравнению с деревом двоичного поиска основание логарифма для B-дерева гораздо больше, что примерно в $log\ t$ раз сокращает количество обращений к диску.

Поиск в B-дереве похож на поиск в двоичном дереве. Разница в том, что в вершине x мы выбираем один вариант из ($n[x]$+1), а не из двух.

Алгоритм 5.2. Поиск в B-дереве

Вход: B-дерево T, r – корень дерева T, k – искомый ключ.

Выход: Упорядоченная пара (y, i), где y – вершина, i – индекс, для которого $key_i[y] = k$ (обозначим через *УП*). Если ключ k не найден, то возвращается *NULL*.

Метод:

```
начало
    выдать ПОИСК_в_В_дереве (r, k).
конец
функция ПОИСК_в_В_дереве (x, k) : УП  | x – вершина дерева T,
                                      | k – искомый ключ
    i ← 0
    пока i < n[x] и k > key_i[x] выполнить
        i ← i + 1                      | поиск ключа k
    конец цикла
    если i < n[x] и k = key_i[x] то    | ключ найден
        выдать (x, i)
    если leaf [x] = истина то          | дошли до листа
        выдать NULL
    иначе
        начало
            Disk_READ ( C_i[x] )       | чтение с диска вершины C_i[x]
                                       | поиск ключа в потомке:
            выдать ПОИСК_в_В_дереве (C_i[x], k)
        конец
конец функции
```

Конец алгоритма 5.2

Функция *ПОИСК_в_В_дереве* проходит в процессе рекурсии вершины от корня в нисходящем порядке. Количество дисковых страниц, которым выполняется обращение, равно $O(h) = O(log_t\ n)$, где h – высота B-дерева, а n – количество вершин в нем. Поскольку $n[x] < 2 \cdot t$, количество итераций цикла **пока** в каждой вершине равно $O(t)$, общее время вычислений равно $O(t \cdot h) = O(t \cdot log_t\ n)$.

5.3.2 Разбиение вершины B-дерева

Добавление элемента в B-дерево – более сложная операция по сравнению с бинарными деревьями. Ключевым местом является разбиение полной вершины с $2t - 1$ ключами на две вершины, имеющие по $t - 1$ ключей в каждой. При этом ключ – медиана $key_t[y]$ – отправляется к родителю x вершины y и становится разделителем двух полученных вершин. Это возможно, если вершина x неполна. Если y – корень, то высота дерева увеличивается на 1.

Продемонстрируем процесс разбиения вершины на примере с рис. 5.9 при $t = 4$. Пусть нам требуется разделить вершину y на две: y и z. Ключ медиана вершины y $key_3[y] = S$ переходит к ее родителю – вершине x. Ключи, большие S, как показано на рисунке, переписываются в новую вершину z, сына вершины x.

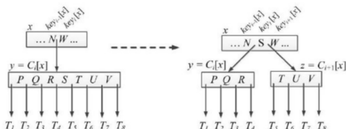


Рис. 5.9. Разбиение вершины с $t = 4$

Ниже приведена процедура *B_tree_Split_Child* из [5], осуществляющая разбиение полной вершины. Входными данными для нее являются неполная внутренняя вершина x, число i и полная вершина y, являющаяся сыном вершины x: $y = C_i[x]$. Будем считать, что x и y находятся уже в оперативной памяти.

процедура *B_tree_Split_Child* (x, i, y)
  $z \leftarrow$ *создать_вершину*()
  $leaf\,[z] \leftarrow leaf\,[y]$
  $n[z] \leftarrow t - 1$ | *– количество ключей в новой вершине z*
  для всех j **от** 0 **до** $t - 2$ **выполнить**
  $key_j\,[z] \leftarrow key_{j+t}\,[y]$ | *запись ключей в вершину z из вершины y:*
  конец цикла | *выбираются ключи, начиная с медианы* $key_t[y]$
  если $leaf\,[y]$ = *ложь* **то**
    для всех j **от** 0 **до** $t - 1$ **выполнить**
      $C_j[z] \leftarrow C_{j+t}[y]$; | *сыновьями вершины z становятся сыновья*
  конец цикла | *вершины y, начиная с* C_t;

```
n[y] ← t – 1               | – количество ключей в вершине y
для всех j от n[x]+1 вниз до i выполнить
    C_{j+1}[x] ← C_j[x]    | у вершины x ссылки на сыновей от позиции i и
конец цикла                | до конца сдвинуты на одну позицию вправо;
C_{i+1}[x] ← z             | i+1 сыном вершины x становится вершина z
для всех j от n[x] вниз до i выполнить
    key_{j+1}[x] ← key_j [x] | у вершины x ключи от позиции i и до конца
конец цикла                 | сдвинуты на одну позицию вправо;
key_i [x] ← key_t [y]       | на i-е место в вершину x записывается ключ
                            | медиана из вершины y;
n[x] ← n[x] + 1             | – количество ключей в вершине x;
Disk_Write (y)              | записать на диск вершины: y, z, x ;
Disk_Write (z)
Disk_Write (x)
конец процедуры
```

Вершина y имела $2 \cdot t$ сыновей, после разбиения в ней осталось t сыновей. Остальные t сыновей стали сыновьями новой вершины z.

5.3.3 Добавление элемента в В-дерево

Процедура *B_tree_insert* (T, k) [5] добавляет элемент k в В-дерево T, пройдя один раз от корня к листу. На это требуется время $O(t \cdot log_t n)$ и $O(h)$ обращений к диску, если высота дерева равна h. По ходу дела с помощью функции *B_tree_Split_Child* разделяются вершины, которые являются полными и которые имеют неполного родителя. В результате выполняется проход до неполного листа, в который и добавляется новый элемент.

```
процедура B_tree_insert (T, k)
    r ← root(T)                       | r – корень дерева T
    если n[ r ] = 2·t – 1 то
        начало                    | добавление в дерево с полным корнем
            s ← создать_вершину()
            root(T) ← s               | s становится корнем
            leaf[ s ] ← 0             | s не является листом
            n[ s ] ← 0                | ключей в s нет
            C_1[ s ] ← r              | у s один сын – корень дерева T
            B_tree_split_child (s, 1, r)  | разбили полную вершину r,
            B_tree_insert_nonfull (s, k) | добавили элемент k в поддерево
        конец                         | с корнем в неполной вершине s;
    иначе                             | добавили элемент k в поддерево
            B_tree_insert_nonfull (r, k)  | с корнем в неполной вершине r
конец процедуры
```

Рекурсивная процедура *B_tree_insert_nonfull* (*r, k*) осуществляет добавление элемента в неполную вершину, при необходимости, выполнив разделение вершины. Если вершина является листом, то ключ *k* в нее добавляется. Иначе ключ *k* добавляется к поддереву, корень которого является сыном этой вершины. Если этот сын – полная вершина, то проводится ее разделение.

```
процедура B_tree_insert_nonfull(x, k)
  i ← n[x];
  если leaf[ x ] = истина то                   | ключ вставляется в лист
    начало
      пока i ≥ 0 и k < key_i[ x ] выполнить
          key_{i+1}[ x ] ← key_i [ x ]           | все ключи, большие k,
          i ← i – 1                            | сдвигаются вправо;
      конец цикла
      key_{i+1}[x] ← k                          | на освободившееся место
                                               | записывается ключ k;
      n[x] ← n[x] + 1      | количество ключей увеличивается
      Disk_WRITE(x)
    конец
  иначе
    начало
      пока i ≥ 0 и (k < key_i[x] выполнить
        i ← i – 1                              | поиск нужного сына
      конец цикла
      i ← i + 1;
      Disk_READ ( C_i (x) )
      если n[ C_i [x] ] = 2·t–1) то  | если сын – полная вершина,
        B_tree_split_child (x, i, C_i[x]) | то ее разделение;
      если (k > key_i [x]) то                   | если ключ k больше ключа
        начало                                 | в вершине x на месте i, то
            i ← i + 1;                         | вставить ключ k в i+1 сына
            B_tree_ insert_nonfull ( C_i[x], k)
        конец |– вторую часть от разделенной полной вершины
    конец
конец процедуры
```

5.3.4 Удаление элемента из В-дерева

Продемонстрируем основные случаи алгоритма удаления элемента из В-дерева на примере из [5], показанном на рис. 5.10. Для этого дерева $t = 3$.

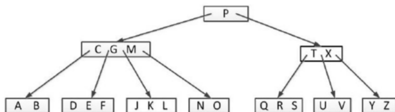


Рис. 5.10. В-дерево для примера

В результате удаления ключа *F* из листа получим дерево на рис. 5.11.

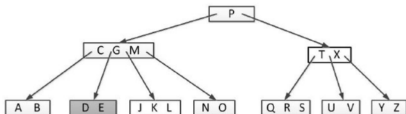


Рис. 5.11. Из В-дерева с рис. 5.10 удален ключ *F*

Из внутренней вершины дерева (рис 5.11), сын которой имеет не менее *t* элементов, удален ключ *M*. Результат этого представлен ниже на рис. 5.12.

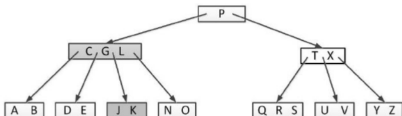


Рис. 5.12. Из В-дерева с рис. 5.11 удален ключ *M*

Удаление ключа *G* из внутренней вершины дерева (рис. 5.12), сыновья которой имеют по t–1 ключу, приводит к их слиянию, как показано на рис. 5.13.

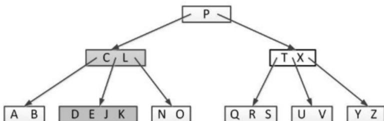


Рис. 5.13. Из B-дерева с рис. 5.12 удален ключ *G*

Удаление ключа *D* в вершине, отцом которой является вершина с количеством ключей меньше *t*, приводит к слиянию промежуточных вершин дерева (рис. 5.14) и в нашем примере к уменьшению высоты дерева.

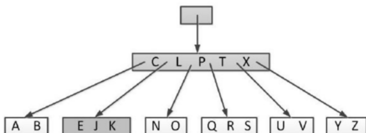


Рис. 5.14. Из B-дерева с рис. 5.13 удален ключ *D*

В результате удаления ключа *C* из вершины дерева с рис. 5.14 ключ из сына этой вершины, имеющий строго больше *t*–1 сыновей, записан в вершину, в которой находился ключ *C*, что и отображено на рис. 5.15.

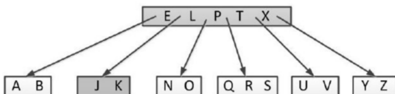


Рис. 5.15. Из B-дерева с рис. 5.14 удален ключ *C*

Описание функций, реализующих удаление ключей из B-дерева, авторы оставляют для самостоятельной работы.

5.4 Дерево отрезков

Дерево отрезков – это структура данных, которая позволяет в массиве на заданном отрезке эффективно реализовать такие операции как нахождение минимального (максимального) элемента, суммы элементов массива, замену i-го элемента или изменение элементов на целом отрезке.

Структура представляет собой дерево, листьями которого являются элементы исходного массива. Другие вершины этого дерева имеют по два сына и содержат результат операции от своих сыновей. Таким образом, корень содержит результат искомой функции от всего массива, левый сын корня содержит результат этой функции на отрезке [0 .. $n/2$], а правый – результат на отрезке [$n/2 + 1$ .. $n - 1$] и т. д.

Отметим, что дерево отрезков содержит $2 \cdot n - 1$ вершин, его высота равна $O(\log n)$.

5.4.1 Создание дерева отрезков

Процесс построения дерева отрезков по заданному массиву a можно выполнить следующим образом. Сначала нужно записать значения элементов $a[i]$ в соответствующие листья дерева. Затем требуется посчитать значения для вершин, которые являются прямыми предками посчитанных вершин как функцию от значений этих вершин и т. д.

Будем хранить дерево отрезков в виде массива так, как описано в разделе 3.1. Пусть нумерация элементов этого массива начинается с нуля. Тогда левый сын вершины i будет расположен в позиции $2 \cdot i + 1$, правый сын – в позиции $2 \cdot i + 2$. Увеличим длину массива так, чтобы она равнялась ближайшей степени двойки. Это сделано, для того чтобы не допустить обращение к несуществующим элементам массива при дальнейшем процессе построения. Пустые элементы необходимо заполнить нейтральными элементами, которые не будут влиять на конечный результат.

Итак, при построении дерева заполним массив, представляющий дерево отрезков, таким образом, чтобы i-й элемент являлся бы значением функции от элементов с номерами $2 \cdot i + 1$ и $2 \cdot i + 2$. Можно выполнить это рекурсивно.

Выделяют два способа построения дерева отрезков: построение снизу и построение сверху. В первом случае дерево строится, начиная от листьев к корню, во втором случае – от корня к листьям, как показано в описанном ниже алгоритме 5.3.

Алгоритм 5.3. Построение дерева отрезков сверху

Вход: Массив a, функция f.
Выход: Дерево отрезков, представленное массивом *tree*.

Метод:

начало

$i \leftarrow 0$

left $\leftarrow 0$

right $\leftarrow n - 1$

tree_ build (*i*, *left*, *right*)

конец

процедура *tree_ build* (*i, left, right*)

если *left* = *right* **то** | элементы массива записаны

tree [*i*] ← *a*[*left*] | в листья дерева

иначе

начало

$m \leftarrow (left + right) / 2$ | середина отрезка

tree_ build ($2 \cdot i + 1$, *left*, *m*)

tree_ build ($2 \cdot i + 2$, $m + 1$, *right*)

tree[*i*] ← *f* (*tree*[$2 \cdot i + 1$], *tree*[$2 \cdot i + 2$])

конец

конец процедуры

Конец алгоритма 5.3

Стоит упомянуть, что в качестве оптимизации операции умножения на 2 и деления на 2 следует заменить битовыми операциями арифметического сдвига. Пример дерева отрезков для запроса суммы показан на рис. 5.16.

5.4.2 Запрос суммы на отрезке

‖ Нахождение суммы на отрезке называется *запросом суммы*.

Имея построенное дерево отрезков, рассмотрим, как реализуется эта операция.

Обработка дерева отрезков начинается с его корня. Далее следует определить, в какой из двух его сыновей попадает отрезок запроса [*l* .. *r*] .

Возможны два варианта:

1) отрезок [*l* .. *r*] является потомком только одного сына;
2) отрезок [*l* .. *r*] является потомком обоих сыновей.

В первом случае необходимо применить алгоритм к сыну, в котором лежит отрезок [*l* .. *r*].

Рассмотрим второй случай. Пусть левый сын представляется отрезком [l_1 .. r_1], а правый – [l_2 .. r_2]. Сначала посчитаем ответ на запрос суммы в границах [*l* .. r_1], а затем – ответ в границах [l_2 .. *r*], эти ответы сложим. Отметим, что $l_2 = r_1 + 1$.

В алгоритме 5.4 используется рекурсивная функция, на вход которой передается информация о текущей вершине дерева: номер вершины *i* и

границы *left* и *right*, а также границы *l* и *r* текущего запроса. На рис. 5.16 показана нумерация вершин.

Алгоритм 5.4. Запрос суммы на отрезке

Вход: Дерево отрезков *tree*, границы отрезка *l* и *r*. Тип элементов массива *tree* – целый.
Выход: Сумма элементов массива с *l*-го по *r*-ый.
Метод:

```
начало
    i ← 0
    left ← 0
    right ← n – 1
    ввести (l, r)
    выдать sum (i, left, right, l, r)
конец

функция sum (i, left, right, l, r) : целое число
    если l > r то
        выдать 0
    если left = l и right = r то
        выдать tree[i]
    m ← (left + right) / 2                    | середина отрезка
    выдать ( sum (2·i + 1, left, m, l, min(r, m)) +
            sum (2·i + 2, m + 1, right, max(l, m + 1), r) )
конец функции
```

Конец алгоритма 5.4

В теле функции *sum* содержится два рекурсивных вызова. Первый из них вычисляет запрос в левом поддереве в границах от *l* до *r,* если запрос содержится в одном этом поддереве, иначе – в границах от *l* от *m,* где *m* – позиция самого крайнего правого элемента поддерева. Аналогично, второй рекурсивный вызов вычисляет запрос суммы в правом поддереве.

Таким образом, вычисление запроса представляет собой спуск по дереву отрезков, используя при этом уже посчитанные суммы по каждому отрезку. Если делать это разбиение правильным образом, то благодаря структуре дерева отрезков число необходимых подотрезков всегда будет $O(\log n)$, что и дает эффективность работы дерева отрезков.

Теперь рассмотрим простую реализацию алгоритма запроса суммы методом *снизу*, которая освобождена от рекурсивных вызовов.

Пусть задан массив $a = [5, 2, 4, 6, 3, 5, 7, 1]$. На рис. 5.16 показано дерево отрезков, листья которого представляют массив *a*, внутренние вершины дерева содержат суммы своих сыновей. Вершины дерева пронумерованы

слева направо и соответствуют позициям в массиве, представляющем дерево отрезков *tree* = [33, 17, 16, 7, 10, 8, 8, 5, 2, 4, 6, 3, 5, 7, 1].

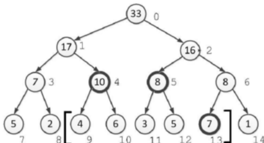


Рис. 5.16. Дерево отрезков

Допустим, требуется просуммировать элементы массива *a*: 4, 6, 3, 5, 7, т. е. *left* = 2, *right* = 6. В дереве отрезков эти элементы находятся во второй половине, поэтому сдвинем границы на *n* – 1, тогда *left* = 9, *right* = 13.

Итак, если левая граница – нечетное число, то нужно подняться вверх к отцу вершины *tree*[*left*]. В нашем примере *tree*[*left*] есть *tree*[9], отцом этой вершины будет вершина *tree*[4], она представляет сумму чисел 4 и 6.

Если же левая граница – четное число, тогда к текущей сумме надо добавить элемент *tree*[*left*] и подняться по дереву к отцу вершины *tree*[*left* + 1], позиция которого в дереве отрезков равна *left* / 2, деление выполняется в целых числах.

Если правая граница – четное число, то нужно подняться вверх к отцу вершины *tree*[*right*], позиция которого в дереве отрезков вычисляется с помощью двух операторов: **`right /= 2`** и **`right--`**.

Если же правая граница – нечетное число, тогда к текущей сумме надо добавить элемент *tree*[*right*]и подняться по дереву к отцу вершины *tree*[*right* – 1].

В примере на рис. 5.16 *right* = 13, поэтому к текущей сумме добавляется число 7. Отцом вершины *tree*[12] является вершина *tree*[5], которая представляет сумму чисел 3 и 5.

Функция, реализующая описанный алгоритм, представлена ниже.

```
int sum(int left, int right)
{
      int s = 0;
      while (left < right)
      {
            if (left % 2 == 0) // левая граница - четное
                  s += tree[left];
            if (right % 2 == 1) //правая граница - нечетное
```

```
            s += tree[right];
            left /= 2;
            right /= 2;
            right--;
      }
      if (left == right)
            s += a[left];
      return s;
}
```

Итак, общую сумму на запрашиваемом отрезке будут представлять вершины дерева *tree*[4] = 10, *tree*[5] = 8 и *tree*[13] = 7, на рис. 5.16 они выделены.

5.4.3 Запрос обновления

Замена *i*-го элемента в дереве отрезков называется запросом обновления. Это более простой запрос по сравнению с запросом подсчета суммы. Он также может быть реализован рекурсивно. Функции, реализовывающей данный запрос, передается текущая вершина дерева отрезков. В теле функции выполняется рекурсивный вызов от того сына, который содержит позицию *i* в своем отрезке. Затем пересчитывается значение суммы в текущей вершине точно таким же образом, как это выполнялось при построении дерева отрезков.

В данном учебном пособии реализация других операций не рассмотрена, предлагается выполнить их в качестве самостоятельной работы.

5.5 Дерево Фенвика

Дерево Фенвика – это структура данных, похожая на дерево отрезков, но более простая в реализации. Впервые оно было описано в 1994 г. Питером Фенвиком. Основные операции, которые можно выполнять с помощью дерева Фенвика, включают:

- вычисление значения некоторой ассоциативной, коммутативной, обратимой функции *f* на отрезке [*l* .. *r*] за время *O* (*log n*);
- изменение значения любого элемента за *O* (*log n*).

Для хранения этого дерева требуется *n* элементов. Оно легко обобщается на случай многомерных массивов. Чаще всего в качестве функций используются сумма чисел на отрезке, произведение, а также при определенной модификации, нахождение максимума, минимума на отрезке. Далее в качестве *f* будем рассматривать функцию суммирования.

Пусть имеется массив чисел *a* [0 .. *n* – 1]. Деревом Фенвика является массив *tree_f* [0 .. *n* – 1], в каждом элементе которого хранится сумма определенных элементов массива *a*, а именно:

$$tree_f[i] = \sum_{j=F(i)}^{i} a[j] = a[F(i)] + a[F(i)+1] + \cdots + a[i],$$

где F – некоторая функция. От ее выбора будет зависеть скорость выполнения основных операций. Требуется такая функция F, при которой эти операции будут выполняться за время $O(\log n)$.

Определим эту функцию: $F(x) = x \& (x + 1)$, где & – операция побитового «и». Рассмотрим двоичную запись числа x и обратим внимание на его младший бит. Если он равен 0, то $F(x) = x$. Иначе число x оканчивается группой из одной или нескольких единиц. В результате замены всех этих единиц на нули получим искомое значение $F(x)$.

Вычисление суммы элементов массива a на отрезке $[0 .. r]$ представлено ниже.

```
int sum (r)
{
      result = 0;
      while (r >= 0)
      {
            result += tree_f[r];
            r = F(r) - 1;
      }
      return result;
}
```

Итак, функция *sum* применяется к дереву Фенвика, продвигаясь по массиву *tree_f*, «прыгая» через отрезки там, где это возможно. Сначала к ответу прибавляется значение суммы на отрезке $[F(r) .. r]$, затем берется сумма на отрезке $[F(F(r) - 1) .. F(r) - 1]$, и так далее, пока не достигнем нуля. Это следует из того, что $tree_f[r] = a[F(r)] + a[F(r) + 1] + \ldots + a[r]$.
Тогда
$sum(r) = (a[0] + a[1] + \ldots + a[F(r) - 1]) + (a[F(r)] + a[F(r) + 1] + \ldots + a[r]) = sum(F(r) - 1) + tree_f[r] = sum(F(F(r) - 1) - 1) + tree_f[F(r) - 1] + tree_f[r] = \ldots$

Указанная сумма расписывается до тех пор, пока значение

$$F(\ldots(F(F(r) - 1) - 1) \ldots)$$

не станет равным нулю. Другими словами, значение $sum(r)$ состоит из сумм элементов массива a на отрезках $[F(r) .. r]$, $[F(F(r) - 1) .. F(r) - 1]$ и так далее. Это значение можно также записать в виде суммы:

$sum(r) = tree_f_{r0} + tree_f_{r1} + tree_f_{r2} + \ldots + tree_f_{rm}$,
где $r_0 = r$, $r_{i+1} = F(r_i) - 1 = (r_i \& (r_i + 1)) - 1$.

Например, так как $F[9] = 8$, то $tree_f_9 = a[8] + a[9]$. Также и $F[11] = 8$, но $tree_f_{11} = a[8] + a[9] + a[10] + a[11]$.

Пример

Рассмотрим процесс вычисления $sum(10)$. В таблице 5.1 вычислены r_i, $F(r_i)$ и r_{i+1}.

Таблица 5.1

Вычисление r_i , $F(r_i)$ и r_{i+1} для *sum*(10)

r_i	$F(r_i)$	$r_{i+1} = F(r_i) - 1$
$10 = 1010_{(2)}$	$1010_{(2)} = 10$	9
$9 = 1001_{(2)}$	$1000_{(2)} = 8$	7
$7 = 111_{(2)}$	$000_{(2)} = 0$	стоп

Следовательно, $sum(10) = t_{10} + t_9 + t_7$. Или то же самое, что суммируются интервалы: $sum(10) = a[\,10\,..\,10\,] + a[\,8\,..\,9\,] + a[\,0\,..\,7\,]$.

С помощью дерева Фенвика задача о горнолыжных соревнованиях, описанная в разделе 5.1, решается более эффективно. Это решение можно посмотреть в разборах задач Всероссийской олимпиады школьников по информатике (2004 г.) [11].

Задача о запросах[2]

Задан массив a размера T. Элементы массива нумеруются с нуля. Изначально все элементы массива равны нулю.

Требуется обработать N запросов одного из двух видов:

- Прибавить к элементам массива с L-ого по R-ый заданное число S ($1 \le S \le 10000$). Формат запроса: `1 L R S`.
- Распечатать K-ый элемент массива. Формат запроса: `2 K`.

В первой строке входного файла заданы числа N и T, следующие N строк содержат запросы в описанном выше формате. Все числа целые, все запросы корректны.

Заданы следующие ограничения величин:

$$1 \le N \le 100000,\ 1 \le T \le 1000000.$$

В этой задаче требуется добавлять величину S ко всем элементам на отрезке от L до R и вычислять значение a_K.

Обозначим через $b_i = (a_i - a_{i-1})$. Будем хранить содержимое массива b, а не оригинального массива a. Чтобы вычислить элемент a_K, нужно найти сумму на начальном отрезке массива b: $a_K = \sum_{i=0}^{K}(a_i - a_{i-1}) = \sum_{i=0}^{K} b_i$. При добавлении величины S к элементам массива a на отрезке содержимое массива b будет изменяться следующим образом: b_L += S и b_{R+1} −= S.

Таким образом, нужно уметь изменять одиночные элементы массива b, а также вычислять сумму на его начальном отрезке. Для этого идеально подходит дерево Фенвика.

Ниже представлена программа, реализующая этот подход. Входные данные считываются из файла *input.txt*, результат работы программы записывается в файл *output.txt*.

---

[2] Задача была на сборах школьников НСО в 2010 г. под названием «Снег», автор задачи – С. Ю. Гатилов

```
#include <stdio.h>
int f[1<<20];         //дерево Фенвика размером 1000000
inline void add(int i, int v)
{
   while (i < 1 << 20)
   {
      f[i] += v;  //добавление к i-му элементу дерева
      i |= i + 1; //сдвиг на следующую позицию
   }
}
inline int get(int i)
{
   int res = 0;
   while (i >= 0)
   {
      res += f[i];          //добавление к вычисляемой сумме
      i = (i & (i + 1)) - 1; //сдвиг на следующую позицию
   }
   return res;
}
int main()
{
   freopen("input.txt", "rt", stdin);
   freopen("output.txt", "wt", stdout);
   int n,_;
   scanf("%d%d", &n, &_);
   for(int i=0;i<n; i++)
   {
      int what;
      scanf("%d", &what);
      if (what==1)
      {          //обработка первого запроса
         int l,r,s;
         scanf("%d%d%d", &l, &r, &s);       // ввод данных
         add(l, s);                         // B[L] += S
         add(r + 1, -s);                    // B[R+1] -= S
      }
      else
      {          //обработка второго запроса
         int t;
         scanf("%d", &t);
         printf("%d\n", get(t));
      }
   }
   return 0;
}
```

Время работы этой программы составляет $O(T)$.

5.6 Кучи (пирамиды)

Куча (*heap*) – структура данных, с помощью которой эффективно выполняются операции добавления и удаления минимального / максимального элемента.

Существует много разновидностей куч: бинарные, *k*-ичные, биномиальные, левацкие, косые (*Skew Heaps*), фибоначчиевы. Все они представляются одним или несколькими деревьями и, как правило, реализуются с помощью массивов.

5.6.1 Бинарные кучи

Бинарная куча или *пирамида* – это почти полное бинарное дерево. Основным свойством этой структуры данных является условие, что элементы в ней организованы таким образом, что приоритет вершины не ниже приоритета каждого из ее сыновей.

Пусть задан массив *a* (рис. 5.17), который будет использоваться для реализации бинарной кучи (пирамиды), и пусть приоритет элемента определяется его значением.

Этот массив характеризуется двумя атрибутами: *length*[*a*] – размер массива и *heap_size*[*a*] – количество элементов пирамиды, содержащихся в массиве *a*. Ни один из элементов, следующих после *a*[*heap_size*[*a*]] не является элементом пирамиды.

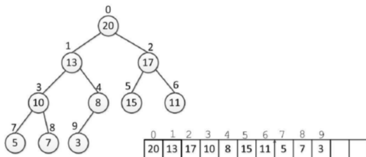


Рис. 5.17. Пирамида и массив, ее представляющий

Различают два вида пирамид: *неубывающие* и *невозрастающие*. В них значения, расположенные в узлах пирамиды, удовлетворяют *свойству пирамиды* [10].

Свойство невозрастающих пирамид (*max-heap property*) заключается в том, что для каждого отличного от корневого узла с индексом *i* выполняется неравенство: $a[parent(i)] \geq a[i]$.

Свойство неубывающих пирамид (*min-heap property*) характеризуется неравенством: $a[parent(i)] \leq a[i]$.

Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева. Значения узлов поддерева такой пирамиды не превышают значения корня этого поддерева. В неубывающей пирамиде в корне дерева находится минимальный элемент. В алгоритме пирамидальной сортировки [10] используются невозрастающие пирамиды. Неубывающие пирамиды часто применяются при использовании очереди с приоритетами, которые рассмотрены в следующем разделе.

Как уже было показано в пункте 3.1, в полном бинарном дереве сыновьями вершины с индексом i являются вершины с индексами $2 \cdot i + 1$ и $2 \cdot i + 2$. В приведенных ниже реализациях функций на языке Си для вершины i вычисляются позиции левого и правого сыновей.

```
int left(int i)                    | левый сын вершины i
{
      return i << 1 + 1;
}

int right(int i)                   | правый сын вершины i
{
      return (i + 1) << 1;
}
```

Отцом вершины с индексом i является вершина с индексом $(i – 1) / 2$, у корневой вершины отца нет.

```
int parent(int i)                  | отец вершины i
{
      return (i - 1) >> 1;
}
```

Высота пирамиды определяется как высота его корня и равна $O(log\ n)$, где n – количество элементов в пирамиде. Время исполнения основных операций в пирамиде пропорционально высоте дерева.

На рис. 5.18 показано, как число 16 добавляется в пирамиду. Находясь в массиве в позиции $i = 10$, оно сравнивается с содержимым ячейки-отца $(i – 1) / 2 = 4$, а затем – с элементом $a[1]$.

Ниже приведено описание функции *Heap_Insert*, реализующей вставку элемента x в пирамиду [5].

процедура *Heap_Insert* (x)
  $i \leftarrow heap_size[a]$
  $a[\ i\ ] \leftarrow x$
  $heap_size[a] \leftarrow heap_size[a] + 1$
  Sift_Up (i)
конец процедуры

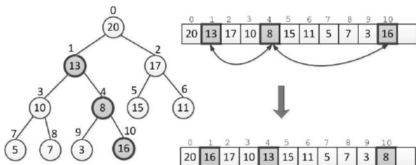


Рис. 5.18. Вставка элемента в пирамиду

Процедура *Heap_Insert* вызывает в процессе своей работы процедуру *Sift_Up* (*i*), которая «просеивает вверх» по пирамиде элемент из вершины с номером *i*, если он не удовлетворяет свойству пирамиды.

процедура *Sift_Up* (*i*)
    если $i = 0$ **то**
        выход
    если $a[\,i\,] \geq a[\,parent(\,i\,)\,]$ **то**
      начало
        $a\,[\,i\,] \leftrightarrow a\,[\,parent\,(\,i\,)]$ | *обмен значениями*
        Sift_Up (*parent* (*i*))
      конец
конец процедуры

Удаление максимального элемента из пирамиды продемонстрировано на рис. 5.19. Второй массив является результатом этого удаления.

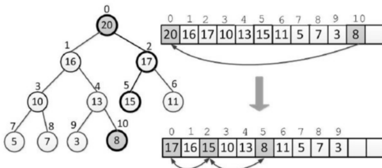


Рис. 5.19. Удаление максимального элемента из пирамиды

На место удаляемого элемента ставится последний элемент пирамиды, который потом «просеивается вниз» по ней для обеспечения сохранения ее свойств.

```
процедура Sift_Down ( a, i )
                        | просеивание элемента с номером i по пирамиде a
    l ← left( i )
    r ← right( i )
    если l < heap_size[ a ] и a[ l ] > a[ i ] то
        largest ← l
    иначе
        largest ← i
    если r < heap_size[ a ] и a[ r] > a[ largest ] то
        largest ← r
    если largest ≠ i то
        начало
            a[ i ] ↔ a[ largest ]          | обмен значениями
            Sift_Down(a, largest)
        конец
конец процедуры

функция Heap_Extract( a ) : ключ
                        | извлечение максимального элемента из кучи
    если heap_size[ a ] < 1 то
        выдать –1                          | куча пуста
    max ← a[ 0 ]
    heap_size[ a ] ← heap_size[ a ] – 1
    a[ 0 ] ← a[ heap_size[ a ] ]
    Sift_Down (a, 0)
    выдать max
конец функции
```

5.6.2 *K*-ичные кучи

В общем случае куча представляет собой одно или несколько деревьев. *k-ичная* куча – это массив, который можно рассматривать как почти полное дерево, где у каждой вершины, кроме, может быть, одной, ровно *k* сыновей.

Например, при $k = 3$, сыновьями вершины с номером *i* будут $k \cdot i + 1$, $k \cdot i + 2$ и $k \cdot i + 3$. Корень имеет номер 0, а отцом вершины с номером *i* является $(i - 1) / k$. На рис. 5.20 показан пример *k-ичной* неубывающей кучи, а также изображен массив, представляющий эту кучу.

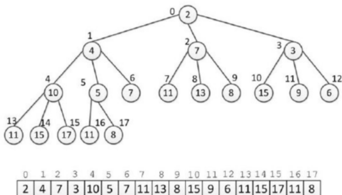


Рис. 5.20. Пример k-ичной кучи

Операция вставки элемента в k-ичную кучу выполняется за время $O\ (log_k\, n)$, а удаление минимального элемента – за время $O\ (k{\cdot}log_k\, n)$.

5.6.3 Фибоначчиевы кучи

Фибоначчиева куча – структура данных, представляющая собой набор деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды. Фибоначчиевы кучи были введены Майклом Фредманом и Робертом Тарьяном в 1984 г. Название рассматриваемых куч связано с использованием чисел Фибоначчи при анализе трудоемкости выполнения операций.

Каждое дерево этой структуры данных удовлетворяет свойству пирамиды. В каждой вершине дерева хранится ключ, а также информация об отце этого узла, о количестве сыновей и ссылка на одного из них, вершины-сыновья объединены в один циклический двусвязный список, каждая вершина содержит указатели на левого и правого соседа.

Фибоначчиевы кучи полезны, если число операций удаления мало по сравнению с остальными операциями. Такая ситуация возникает во многих приложениях. Однако с практической точки зрения программная сложность реализации и высокие значения постоянных множителей в формулах времени работы существенно снижают эффективность их применения, интерес к фибоначчиевым пирамидам в первую очередь сугубо теоретический.

Операции, в которых не требуется удаление, имеют амортизированное (среднее) время работы, равное O (1). Кроме стандартных операций вставки и удаления фибоначчиева куча позволяет за время $O(1)$ выполнять операцию слияния двух куч.

5.7 Очередь с приоритетами

Очередь с приоритетами – часто используемая структура данных, позволяющая хранить пары (k, x) (ключ, значение) и поддерживающая операции добавления пары в некоторое множество S, поиска пары с минимальным/максимальным ключом в этом множестве и извлечения пары с минимальным/максимальным ключом.

Как и пирамиды, очереди с приоритетом бывают невозрастающие и неубывающие. Рассмотрим реализацию невозрастающей очереди с приоритетами [5]. В разных реализациях названия операций могут отличаться.

Поддерживаются следующие операции:

Insert (S, x) – вставка элемента x в множество S, эту операцию можно записать как $S \leftarrow S \cup \{x\}$;

Maximum (S) – выборка элемента с максимальным ключом из множества S;

Extract_Max (S) – выборка элемента с максимальным ключом и удаление его из множества S;

Increase_Key (S, x, k) – увеличение значения ключа, соответствующего элементу x. Значение ключа элемента x заменяется на значение k. Предполагается, что величина k не меньше текущего ключа элемента x.

В неубывающей очереди с приоритетами поддерживаются операции *Insert*(S, x), *Minimum* (S), *Extract_Min* (S), *Decrease_Key* (S, x, k).

Рассмотрим следующий пример. Пусть в цехе некоторые детали поступают на обработку, при этом каждая из них имеет некоторое описание – ключ. Если значение ключа соответствует времени поступления детали на обработку, то минимальное значение ключа имеет та деталь, которая поступила раньше. Для моделирования процесса обработки в этом случае можно использовать структуру данных *очередь*. Если же минимальное значение ключа имеется у детали, которая поступила позже, то при ее обработке можно использовать структуру данных *стек*. И, наконец, если минимальное значение ключа приписано детали, которая меньше всего должна ждать, тогда использование *очереди с приоритетами* приведет к желанному результату.

Невозрастающую очередь с приоритетами можно реализовать с помощью пирамиды. Без ограничения общности будем считать, что очередь с приоритетами реализуется одномерным целочисленным массивом, элементы массива представляют ключи.

Функция *Heap_Maximum*, описание которой приведено ниже, реализует выполнение операции *Maximum* за время $O(1)$.

```
функция Heap_Maximum(a) : ключ
    выдать a[0]
конец функции
```

Функция *Heap_Extract,* описанная в 5.6.1 реализует операцию *Extract_Max*. Время ее работы равно *O* (*log n*).

Функция *Heap_Increase_Key* реализует операцию *Increase_Key*. Элемент очереди с приоритетами, ключ которого подлежит увеличению, идентифицируется в массиве с помощью индекса *i*.

```
процедура Heap_Increase_Key (a, i, key)
    если key < a[ i ] то                 | новый ключ меньше текущего
          выход
    a[ i ] ← key                         | обновление ключа
    пока i > 0 и a[ parent( i ) ] < a[ i ] выполнить
       a[ i ] ↔ a[ parent( i ) ]         | обмен значениями
       i ← parent( i )
    конец цикла
конец процедуры
```

В этой функции элемент *a*[*i*] получает новое значение *key*. После этого выполняется поиск места для него. Если оказывается, что значение текущего элемента превышает значение родительского, то происходит их обмен, и функция продолжает свою работу на более высоком уровне. В противном случае работа прекращается, так как свойство невозрастающих пирамид восстановлено.

На рис. 5.21 приведен пример [5] работы функции *Heap_Increase_Key*. Дерево на рис. 5.21*а* представляет невозрастающую пирамиду, в которой будет увеличен ключ узла, помеченный индексом *i*. Дерево на рис. 5.21*б* – та же пирамида, в которой ключ выделенного узла увеличен до 15.

Далее обрабатываемая пирамида (рис. 5.21*в*) изображена после первой итерации цикла *while*, при этом текущий и родительский по отношению к нему узлы обменялись значениями, и пометка-индекс *i* перешла к родительскому узлу. И, наконец, показана эта же пирамида (рис. 5.21*г*) после еще одной итерации цикла, после которой условие невозрастающих пирамид выполняется, и функция завершает работу.

Время работы этой функции равно *O* (*log n*). Это объясняется тем, что длина пути от обновляемого элемента до корня равна *O* (*log n*).

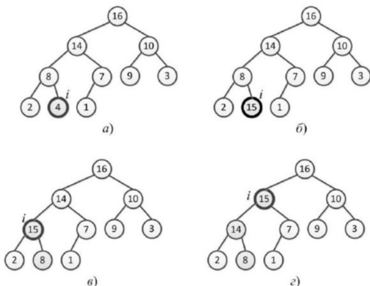


Рис. 5.21. Работа функции *Heap_Increase_Key*

процедура *Max_Heap_Insert*(*a*, *key*)
  a[*heap_size*[*a*]] ← *MINVALUE* | *минимально возможное значение*
  heap_size[*a*] ← *heap_size*[*a*] + 1
  Heap_Increase_Key (*a*, *heap_size* [*a*] – 1, *key*)
конец процедуры

Время вставки в *n*-элементную пирамиду с помощью процедуры *Max_Heap_ Insert* составляет *O* (*log n*). Таким образом в пирамиде время выполнения всех операций по обслуживанию очереди с приоритетами равно *O* (*log n*).

5.8 Декартовы деревья

Декартово дерево – это бинарное дерево, в вершинах которого хранятся пары (x, y), где x – это ключ, а y – это приоритет, таким образом, что по ключам оно является деревом поиска, а по приоритетам – пирамидой.

Предположим, что в декартовом дереве все x и y являются различными. Тогда если некоторая вершина содержит (x_0, y_0), то у всех вершин в левом поддереве значение ключа $x < x_0$, у всех вершин в правом поддереве значение ключа $x > x_0$, а также в вершинах левого и правого поддеревьев выполняется неравенство: $y < y_0$.

Используются также следующие названия, являющиеся синонимом декартова дерева:

Treap = ***tr**ee + h**eap***
Дуча = ***д**ерево + ку**ча***
Дерамида = ***де**рево + пи**рамида***

Дерамиды были предложены Сиделем (*Siedel*) и Арагоном (*Aragon*) в 1996 г.

Декартово дерево можно нарисовать на плоскости, при этом пары (ключ, приоритет) являются соответствующими координатами точек. Пример показан на рис. 5.22, слева – в стандартной нотации дерева, справа – на декартовой плоскости.

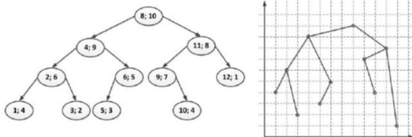


Рис. 5.22. Декартово дерево и его изображение на декартовой плоскости

Заметим, что из заданной последовательности пар (ключ, приоритет) можно построить единственное, вне зависимости от порядка поступления ключей, декартово дерево. Это справедливо, поскольку корнем дерева является элемент с наибольшим приоритетом. В левом поддереве располагаются элементы, ключи которых не превышают ключа корня, в правом поддереве – элементы с большими ключами, при этом дерево продолжает оставаться деревом двоичного поиска. Далее рассматривая рекурсивно, левое и правое поддеревья, корни которых также имеют максимальные приоритеты, приходим к выводу, что положение каждого элемента дерева задано однозначно.

Теорема 5.3

В декартовом дереве из n вершин, приоритеты у которого являются случайными величинами с равномерным распределением, средняя глубина вершины равна $O(log_2 n)$.

Доказательство: доказательство теоремы можно найти в [12].

Операции в декартовом дереве

В декартовом дереве определены две основные операции: *Split* и *Merge.*

Split (*T, k*) – позволяет разрезать декартово дерево *T* по ключу *k*. Результатом операции являются два других декартовых дерева: T_1 и T_2, причем в T_1 находятся все ключи дерева *T*, не большие ключа *k*, а в T_2 – большие ключа *k* (рис. 5.23).

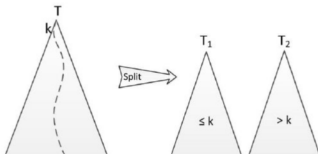


Рис. 5.23. Применение операции *Split* к декартову дереву

Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему либо равному ключу корня. Тогда корень и левое поддерево дерева *T* станут корнем и левым поддеревом дерева T_1. Для нахождения правого поддерева дерева T_1, нужно разрезать правое поддерево дерева *T* на поддеревья T_1^R иT_2^R по ключу *k* и в качестве результата взять поддерево T_1^R. Дерево T_2 совпадет с T_2^R. Это разделение продемонстрировано на рис. 5.24 и описано в алгоритме 5.5. Случай, в котором требуется разрезать дерево по ключу, меньшему ключа в корне дерева *T*, рассматривается симметрично.

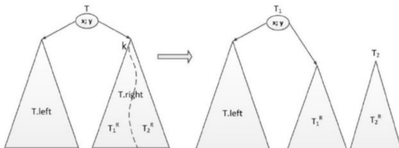


Рис. 5.24. Разделение дерева по ключу *k*, где $k \geq x$

В описании алгоритмов в этом разделе будем считать, что декартово дерево имеет представление, аналогичное тому, как описано в разделе 3.3.3. Отличие состоит в том, что вместо строк в полях x и y будем хранить, соответственно, ключ и приоритет вершины дерева.

Алгоритм 5.5. Реализация операции *Split* (T, k)

Вход: Декартово дерево T, ключ k.
Выход: Декартовые деревья T_1 и T_2, T_1 содержит ключи, не большие k, а T_2 – большие k.
Метод:

```
начало
    Split (T, k) → {T1, T2}
конец

функция Split (T, k) : {декартово дерево, декартово дерево}
    если T = ∅ то
        начало
            T1 ← ∅
            T2 ← ∅
        конец
    иначе
        если k ≥ T.x то
            начало
                Split (T.right, k) → { T.right, T2 }
                T1 ← T
                Recalc (T1)   | перевычисление дополнительных значений
            конец
        иначе
            начало
                Split (T.left, k) →{ T1, T.left }
                T2 ← T
                Recalc (T2)   | перевычисление дополнительных значений
            конец
    выдать {T1, T2}
конец функции
```

Конец алгоритма 5.5

Алгоритм 5.5 содержит рекурсивный вызов функции *Split* для поддерева, высота которого меньше хотя бы на единицу заданного дерева. Поэтому время работы алгоритма равно $O(h)$, где h – высота дерева. Процедуру *Recalc* и перевычисление дополнительных значений рассмотрим позже.

Merge (T_1, T_2) – операция, выполняющая слияние двух декартовых деревьев в одно дерево. Для выполнения этой операции необходимо, чтобы все ключи в первом дереве были меньше ключей второго дерева. В результате получается дерево, в котором есть все ключи из первого и второго деревьев. Корнем дерева T станет вершина с наибольшим приоритетом y, это будет либо корень T_1, либо корень T_2.

Пусть приоритет корня дерева T_1 больше приоритета корня дерева T_2. Тогда корень дерева T_1 станет корнем дерева T, левое поддерево дерева T совпадет с левым поддеревом дерева T_1. Правое поддерево дерева T будет состоять из объединения правого поддерева дерева T_1 и дерева T_2, как показано на рис. 5.25. Симметричный случай, когда приоритет в корне дерева T_2 выше приоритета корня дерева T_1, разбирается аналогично.

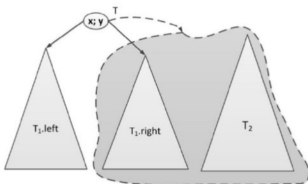


Рис. 5.25. Слияние двух декартовых деревьев

Алгоритм 5.6. Реализация операции *Merge* (T_1, T_2)

Вход: декартовы деревья T_1, T_2.
Выход: декартово дерево T, содержащее ключи из деревьев T_1 и T_2.
Метод:

```
начало
    Merge (T1, T2) → T
конец

функция Merge (T1, T2) : декартово дерево
    если T1 = ∅ то
        T ← T2
    иначе
        если T2 = ∅
            T ← T1
        иначе
```

если $T_1.y > T_2.y$ **то**
начало
Merge $(T_1.right, T_2) \rightarrow T_1.right$
$T \leftarrow T_1$
конец
иначе
начало
Merge $(T_1, T_2.left) \rightarrow T_2.left$
$T \leftarrow T_2$
конец
Recalc (T) | перевычисление дополнительных значений
выдать T
конец функции
Конец алгоритма 5.6

Время работы алгоритма 5.6 равно $O(h)$.

Insert $(T, (k, p))$ – операция добавления в дерево T нового элемента с ключом k и приоритетом p. Этот элемент можно рассматривать как декартово дерево, состоящее из одной вершины q, в котором $q.x = k$, $q.y = p$. Для того чтобы его добавить в дерево T, очевидно, нужно их слить. Но дерево T может содержать ключи как меньшие, так и большие ключа k, поэтому сначала нужно разрезать дерево T по ключу k.

Ниже представлен алгоритм выполнения операции *Insert* $(T, (k, p))$.

Алгоритм 5.7. Реализация I операции *Insert*

Вход: Декартово дерево T, ключ k, приоритет p.
Выход: Декартово дерево T с новой вершиной, содержащей ключ k и приоритет p.
Метод:
Шаг 1. *Split* $(T, k) \rightarrow \{ T_1, T_2 \}$
Шаг 2. Сформировать дерево, состоящее из одной вершины q с ключом k и приоритетом p.
Шаг 3. *Merge* $(T_1, q) \rightarrow T_1$
Шаг 4. *Merge* $(T_1, T_2) \rightarrow T$
Конец алгоритма 5.7

На рис. 5.26 схематично показаны действия алгоритма 5.7.

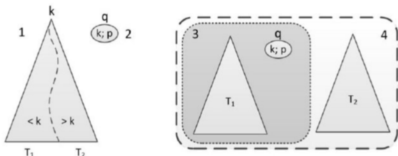


Рис. 5.26. Реализация I операции *Insert* (*T, q*)

Рассмотрим еще одну реализацию операции *Insert.* Напомним, что при вставке ключа в дерево двоичного поиска требовалось идти по дереву вниз, сравнивая этот ключ с ключом текущей вершины и выбирая каждый раз путь влево или вправо, пока не найдем подходящее место. В нашем случае применить напрямую этот алгоритм невозможно, так как можно нарушить условие того, что по приоритетам дерево должно удовлетворять определению пирамиды. Поэтому при спуске по дереву остановимся на первой вершине, в которой значение приоритета оказалось меньше p, обозначим ее за r. После этого разрежем дерево T от найденного ключа $r.x$, получим деревья T_1 и T_2, сформируем новое дерево с корнем в вершине q с поддеревьями T_1 и T_2. Заменим в дереве T поддерево с корнем r на поддерево с корнем q.

Ниже приведен алгоритм этой реализации операции *Insert.*

Алгоритм 5.8. Реализация II операции Insert

Вход: Декартово дерево T, ключ k, приоритет p.

Выход: Декартово дерево T с новой вершиной q, содержащей, ключ k и приоритет p.

Метод:

Шаг 1. Спуститься по дереву T, как по обычному бинарному дереву поиска по ключам до тех пор, пока не встретится первая вершина r, в которой значение приоритета меньше p.
Пусть T_r – поддерево с корнем r.

Шаг 2. *Split* $(T_r, k) \to \{T_1, T_2\}$

Шаг 3. Сформировать вершину q, содержащую ключ k и приоритет p. Деревья T_1 и T_2 записать в качестве левого и правого поддеревьев вершины q.

Шаг 4. Заменить в дереве T поддерево с корнем r на поддерево с корнем q (рис. 5.27).

Конец алгоритма 5.8

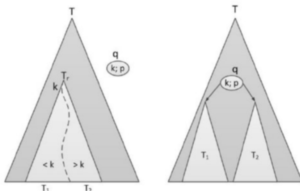


Рис. 5.27. Реализация II операции *Insert* (T,q)

На рис. 5.27 в дереве слева выделено поддерево с корнем в вершине r и два поддерева T_1 и T_2, полученные в результате разбиения по ключу k. На дереве справа отображен результат работы шагов алгоритма 3 и 4.

Remove(T, k) – операция удаления из дерева T элемента с ключом k. Пусть ключи будут целыми числами. Разрежем дерево T на два дерева T_1 и T_2 по ключу k. Тогда все элементы в дереве T_1 будут иметь ключи, меньшие либо равные k. Разрежем дерево T_1 по ключу $k-1$, получим два дерева T_3 и T_4, дерево T_4 будет содержать элементы, ключи которых больше $k-1$, а в T_3 останутся элементы с ключами, меньшими k. Таким образом, после объединения T_3 и T_2 мы получим декартово дерево, не содержащее элементов с ключами k. Реализация этой операции представлена в алгоритме 5.9 и схематично показана на рис. 5.28.

Алгоритм 5.9. Реализация I операции *Remove*

Вход: Декартово дерево T, ключ k.
Выход: Декартово дерево T, в котором удалены вершины с ключом k.
Метод:

Шаг 1. *Split* $(T, k) \rightarrow \{ T_1, T_2 \}$
Шаг 2. *Split* $(T_1, k-1) \rightarrow \{ T_3, T_4 \}$
Шаг 3. *Merge* $(T_3, T_2) \rightarrow T$

Конец алгоритма 5.9

На рис. 5.28 первый шаг алгоритма отображен на дереве слева, оно разбивается на два дерева T_1 и T_2, причем ключи в первом дереве меньше ключей во втором дереве. На втором слева дереве показано разбиение дерева T_1 по ключу $k-1$. Пунктирной линией обведены деревья, которые будут слиты на третьем шаге алгоритма.

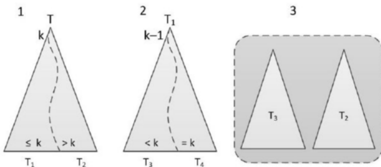


Рис. 5.28. Реализация I операции *Remove* (*T*, *k*)

Рассмотрим еще одну реализацию операции *Remove*, которая описана в алгоритме 5.10.

Алгоритм 5.10. Реализация II операции *Remove*

Вход: Декартово дерево *T*, ключ *k*.
Выход: Декартово дерево *T*, в котором удалена вершина с ключом *k*.
Метод:

Шаг 1. Найти вершину *q*, в которой *q.x* = *k*
Шаг 2. *Merge* (*q.left*, *q.right*) $\to T'$
Шаг 3. Заменить в дереве *T* поддерево с корнем *q* на *T′*

Конец алгоритма 5.10

На рис. 5.29 отображен результат работы трех шагов алгоритма 5.10. В первом дереве слева изображена вершина *q* с ключом *k*, пунктирной линией обведены деревья, которые сливаются на втором шаге, и, наконец, результат слияния – дерево *T′*, вставляется на место вершины *q*.

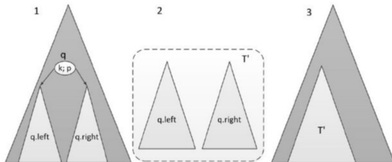


Рис. 5.29. Реализация II операции *Remove* (*T*, *k*)

5.8.1 Построение декартова дерева

Пусть ключи поступают в возрастающем порядке. Ниже предложены два способа построения декартова дерева.

Способ 1

Рассмотрим приоритеты $y_1, y_2, ..., y_n$, выберем среди них максимальный. Пусть это будет y_k. По свойству пирамиды в корне декартова дерева должен быть элемент с максимальным приоритетом. Поэтому вершину (x_k, y_k) объявим корнем дерева. Определив максимальные приоритеты последовательностей $y_1, y_2, ..., y_{k-1}$ и $y_{k+1}, ... , y_n$, получим, соответственно, левого и правого сына.

Время работы этого рекурсивного алгоритма равно $O(n^2)$.

Способ 2

Будем строить дерево слева направо, то есть, начиная с вершины (x_1, y_1) и заканчивая вершиной (x_n, y_n). Пусть последним добавленным элементом будет вершина (x_k, y_k). В дереве она будет самой правой, так как содержит максимальный ключ, поскольку по ключам декартово дерево представляет собой двоичное дерево поиска.

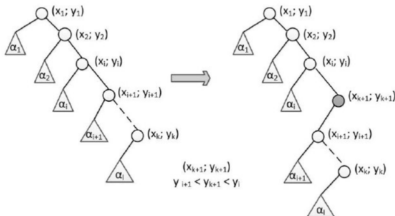


Рис. 5.30. Второй способ построения декартова дерева

При добавлении вершины (x_{k+1}, y_{k+1}), пытаемся сделать ее правым сыном вершины (x_k, y_k). Это возможно при $y_k > y_{k+1}$. Иначе нужно найти место для этой добавляемой вершины. Заметим, что у нее наибольший ключ, так что в конечном итоге она будет самой правой вершиной дерева. Поэтому, искать место для ее вставки нужно в самой правой ветке дерева. Для этого вернемся к предку последнего элемента (x_k, y_k) и сравним значение его приоритета с y_{k+1} и т. д. Поднимаемся к предкам до тех пор, пока приоритет в

рассматриваемой вершине меньше приоритета добавляемой вершины. Пусть это будет вершина (x_i, y_i) (см. рис. 5.30). После этого делаем вершину (x_{k+1}, y_{k+1}) ее правым сыном, а предыдущего правого сына, вершину(x_{i+1}, y_{i+1}), делаем левым сыном вершины (x_{k+1}, y_{k+1}).

Таким образом, каждую вершину мы посетим максимум дважды: при непосредственном добавлении и поднимаясь вверх по предкам. Из этого следует, что построение декартова дерева происходит за время O(n).

5.8.2 Использование декартова дерева

В декартовом дереве, как и в любом дереве поиска, можно искать ключ, добавлять и удалять его за логарифмическое время.

На примере решения нескольких задач продемонстрируем, как можно, еще использовать декартово дерево.

Задача поиска порядковых статистик

k-я *порядковая статистика* – это элемент множества, который стоял бы на *k*-м месте, если бы мы отсортировали это множество элементов по возрастанию.

Задача состоит в определении *k*-го по величине элемента за время $O(\log n)$.

Пусть у нас имеется декартово дерево. Требуется быстро находить в нем *k*-й по порядку возрастания ключ. Если представить дерево в виде отсортированного массива, то элемент с индексом *k*, будет ответом на нашу задачу (рис. 5.31).

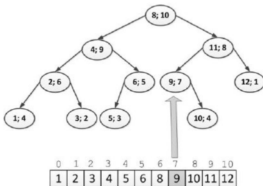


Рис. 5.31. Представление декартова дерева в виде массива

Для решения этой задачи в поле *size* каждой вершины будем хранить размер поддерева, корнем которого является данная вершина, как показано на рис. 5.32. Использую эту информацию можно определить, в каком поддереве будет располагаться нужный элемент.

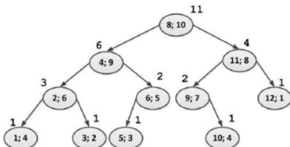


Рис. 5.32. Декартово дерево с информацией о размерах поддеревьев

Алгоритм поиска k-й статистики состоит из следующих действий. Число k сравнивается с размером левого поддерева, который хранится в переменной $size_L$. Если $size_L = k$, то искомым элементом является корень дерева. Если $size_L > k$, то искомый элемент находится где-то в левом поддереве, поэтому процесс поиска повторяется в левом поддереве. Иначе искомый элемент находится в правом поддереве. Чтобы корректно реагировать на размеры поддеревьев справа, число k уменьшается на $size_L + 1$, и процесс поиска повторяется для правого поддерева.

Промоделируем этот поиск для примера, показанного на рис. 5.31 для числа $k = 7$. Для корня этого дерева $size_L = 6$. Так как $k = 7$, направим поиск в правое поддерево, уменьшая при этом k на (6+1) единиц. Для корня правого поддерева (11; 8) $size_L = 2$. И, так как $k = 0$, направляем поиск в левое поддерево, корнем которого является вершина (9; 7). У вершины (9; 7) нет левого поддерева, $size_L = 0 = k$, следовательно, ответ найден – это вершина (9;7).

Пересчет размеров поддеревьев

До сих пор мы не касались вопроса пересчета размеров деревьев после добавления/удаления ключа. Эти действия выполняются, используя операции *Split* и *Merge*. Если последние реализовать таким образом, чтобы они поддерживали размеры поддеревьев, то операции вставки и удаления ключа автоматически будут выполняться корректно. При этом в операции создания дерева, состоящего из одной вершины, нужно установить размер дерева, равный 1.

Напомним, что в дополнительном поле *size* для каждой вершины хранится информация о размере поддерева, корнем которого является данная вершина. В реализации операции *Merge* (алгоритм 5.6) выбирается корень для нового дерева, а затем рекурсивно сливается одно из его поддеревьев с другим деревом.

Рассмотрим случай, когда сливать нужно правое поддерево. Пусть после выполнения *Merge* на поддеревьях все размеры вычислены верно. Тогда осталось посчитать размер только в самом корне нового дерева: $size \leftarrow T.left.size + T.right.size + 1$. В алгоритме 5.6 вызов процедуры *Recalc*(T) выполняет этот пересчет размеров дерева.

Аналогичная ситуация в реализации операции *Split* (алгоритм 5.5). В ней в зависимости от значения ключа в корне дерева рекурсивно делится по указанному ключу либо левое, либо правое поддерево. Предположим, что делится правое поддерево: *T.right*. Пусть в рекурсивных вызовах *Split* все размеры поддеревьев определены верно. Тогда перед завершением функции нужно посчитать значение в корне будущего дерева T_1: *Recalc*(T_1).

Задача о запросе суммы на отрезке

В дополнительном поле *sum* каждой вершины будем хранить сумму значений ключей, расположенных в поддереве, корнем которого является эта вершина.

Чтобы ответить на запрос суммы на произвольном отрезке $[a .. b]$, надо выделить из дерева часть, соответствующую этому отрезку. Нетрудно понять, что для этого достаточно сначала применить операцию *Split* (T, a), получив при этом деревья T_1 и T_2.

Затем нужно применить операцию *Split* ($T_2, b - a + 1$) к дереву T_2, в результате получить деревья T_3 и T_4. Дерево T_3 будет состоять из элементов в отрезке $[a .. b]$. Следовательно, ответ на запрос будет находиться в поле *sum* корня дерева T_3.

Задача о нахождении максимума на отрезке

Зачастую требуется хранить в дереве помимо ключей и другие данные, с которыми приходится производить какие-то манипуляции. Например, пусть на вход постоянно поступают, а порою удаляются, ключи x, и с каждым из них связана соответствующая цена, записанная в дополнительном поле *cost*. Необходимо поддерживать быстрые запросы на максимум цены на множестве таких элементов, где $a \le x < b$.

Для решения этой задачи в дополнительном поле *max* для каждой вершины будем хранить максимум цены элементов, расположенных в поддереве, корнем которого является эта вершина. Это легко выполнить, и пусть это реализуется с помощью функции *MaxCostOf* (T).

Тогда для нахождения максимума на отрезке нужно применить операцию *Split* ($T, a - 1$), получив два дерева T_1 и T_2. Затем снова применить эту операцию к дереву T_2, содержащему ключи, большие либо равные a: *Split*(T_2, b), результат – деревья T_3 и T_2. Дерево T_3 будет содержать элементы, у которых ключи принадлежат искомому интервалу. И наконец, применить функцию *MaxCostOf* (T_3).

Поддержка множественных операций

Во многих задачах в процессе каких-то манипуляций востребовано изменение пользовательской информации. Например, требуется изменить значение *cost* в одном заданном элементе. Тогда для решения задачи о нахождении максимума в некоторых вершинах дерева нужно перевычислить значения *max*. Для этого, двигаясь от корня дерева вниз, находится нужная вершина, и значение *cost* в ней изменяется. Затем, двигаясь обратно снизу вверх, пересчитываются значения параметров *max*. Перевычисление значений производится только в тех вершинах, которые посещались по пути от корня к вершине.

Декартово дерево позволяет также производить групповую модификацию элементов: умножение на число, замена значения, добавления некоторой величины к значениям, установка каких-либо свойств, в том числе на заданном отрезке.

Пусть, например, к каждому значению *cost* в дереве (или поддереве) нужно прибавить какое-то одно и то же число a. Для решения этой задачи заведем в каждой вершине дополнительное поле *add*. Значение a, записанное в этом поле, будет указывать на то, что всем вершинам поддерева, корнем которого является данная вершина, требуется добавить число a. Обратим внимание, что *требуется* добавить, но пока мы не будем реализовывать это добавление. На него можно посмотреть как на действие, выполнение которого отложено на некоторое время.

Тогда реальная цена корня равна $cost\ (T) = T.cost + T.add$.

Аналогичным способом определяется сумма цен в дереве T:

$$sum(T) = T.sum + T.add \cdot T.size.$$

Таким образом, для получения правильного значения суммы с учетом всех отложенных операций к *sum* нужно прибавить значение *add*, умноженное на число *size* – количество элементов в поддереве.

Рассмотрим вопрос, когда же потребуется добавить вершинам поддеревьев число a, и что при этом должны поддерживать операции *Split* и *Merge*. Предположим, что при своем вызове операция *Split* делит правое поддерево *T.right*. Тогда для решения нашей задачи нужно выполнить следующую последовательность действий:

1) В корне дерева добавить к значению *cost* значение, записанное в поле *add*:

$$T.cost \leftarrow T.cost + T.add.$$

2) Поскольку при вызове *Split* левое поддерево не изменяется, то отложить выполнение добавления константы для всех его вершин, для этого записать в корень левого поддерева:

$$T.left.add \leftarrow T.left.add + T.add.$$

Аналогично, записать в корень правого поддерева:

$$T.right.add \leftarrow T.right.add + T.add.$$

Обе эти операции следует выполнить вызова операции *Split* на правом поддереве.
3) Применить *Split* к правому поддереву. Результатом работы данной операции будут два корректных декартовых дерева.
4) Поскольку в корне мы добавили константу, а для потомков это добавление взяли себе на заметку, то значение *add* в корне дерева надо обнулить:

$$T.add \leftarrow 0.$$

Итак, реальные обновления проводятся лишь в тех вершинах декартова дерева, которые затрагиваются операцией *Split*. Для остальных вершин устанавливается пометка, что константа должна к ним добавиться позже.

Аналогичные рассуждения можно провести и с операцией *Merge*.

Предположим, что применение операции *Merge* приводит к объединению правого поддерева $T_1.right$ с деревом T_2 (рис. 5.25), тогда выполняется следующая последовательность действий:

1) В корне дерева T_1 добавить к значению *cost* значение, записанное в поле *add*:
$$T_1.cost \leftarrow T_1.cost + T_1.add.$$
2) Отложить выполнение добавления константы для обоих поддеревьев дерева T_1:
$$T_1.left.add \leftarrow T_1.left.add + T_1.add$$
$$T_1.right.add \leftarrow T_1.right.add + T_1.add$$
3) Поскольку в левом и правом поддеревьях установлена информация о добавлении константы, а в корне это добавление выполнено, то значение *add* в корне дерева надо обнулить: $T_1.add \leftarrow 0.$
4) Применить операцию *Merge* ($T_1.right$, T_2), результатом работы которой будет декартово дерево, в некоторых вершинах в поле *add* этого дерева будет иметься указание о добавлении константы.

Следует не забывать после всех операций в корнях новых деревьев вызывать соответствующий вариант функции *Recalc* (см. алгоритмы 5.5 и 5.6), а все необходимые вычисления внести в тело этой функции.

Отметим, что с помощью декартовых деревьев можно решать задачи, в которых требуется сбалансированное дерево поиска, например, эффективная реализация множеств или словарей, задачи с запросами на диапазонах.

Рассмотрение декартова дерева по *неявному* ключу выходит за рамки этого учебного пособия.

5.9 Система непересекающихся множеств (СНМ)

Система непересекающихся множеств – это структура данных, которая реализует разбиение множества. Каждое подмножество, входящее в разбиение, характеризуется своим представителем.

Это понятие было введено Тарьяном (*Tarjan*) в 1975 г.

СНМ поддерживает следующие операции:

MakeSet (x) – добавление в СНМ нового элемента x, который заносится в новое подмножество, x становится представителем этого подмножества.

FindSet (x) – поиск подмножества, которому принадлежит элемент x, и определение его представителя.

Union (x, y) – объединение в одно множество двух подмножеств, к которым принадлежат элементы x и y. Результатом работы является элемент, который становится представителем этого множества.

Рассмотрим некоторые подходы, которые можно было бы использовать при реализации СНМ.

5.9.1 Простая реализация

В этой реализации для каждого элемента множества хранится номер или другими словами цвет подмножества, к которому этот элемент принадлежит. На рис. 5.33 показано пять подмножеств, их нумерация указана сверху. Два множества с номерами 2 и 3 объединены в одно множество с номером 2.

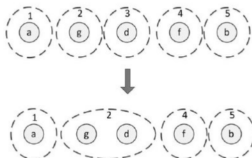


Рис. 5.33. Объединение двух множеств с номерами 2 и 3 в одно множество

Если хранить множества, например, в виде массива, то при таком подходе операция *FindSet* (x) выполняется за $O(1)$, а операция *Union* (x, y) – за $O(|V|)$. Последняя оценка не удовлетворяет требованию СНМ.

5.9.2 Реализация с помощью списков

1 *способ*. Если хранить множества в виде линейных списков с указателями на начало и конец списка, и в качестве представителя множества возвращать голову списка, то операция *Union* (x, y), слияние двух списков, выполняется за $O(1)$, а *FindSet* (x), поиск элемента в списке – за O ($|V|$).

2 *способ*. Каждый элемент списка может содержать ссылки на следующий элемент и на первый элемент списка. Кроме того, для каждого списка хранятся указатели на его первый и последний элементы. При такой реализации операция *FindSet* (x) требует времени O (1). При выполнении операции *Union* (x, y) список, содержащий элемент y, добавляется к концу списка, содержащего элемент x. При этом требуется установить правильные указатели на начало списка для всех бывших элементов множества, содержащего y. Время на выполнение операции *Union* (x, y) линейно зависит от размера множества, которому принадлежит y, т. е. составляет O ($|V|$).

5.9.3 Весовая эвристика

Весовая эвристика – это улучшение простой реализации, в которой следует перекрашивать элементы из множества меньшей мощности, как показано на рис. 5.34. В этой реализации для каждого множества из СНМ необходимо хранить его мощность.

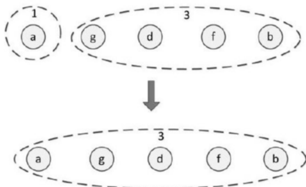


Рис. 5.34. При объединении элементы из меньшего множества перекрашиваются

При каждом объединении, множество, в котором оказывается объект, увеличивается не более чем вдвое. Тогда после первого объединения элемент содержится во множестве, в котором не более двух элементов, после второго – четырех, и так далее. В силу того, что множество не может содержать более n элементов, количество объединений не превосходит O ($log\ n$).

Если множества реализуются в виде списков вторым способом, то вместе с каждым списком нужно хранить информацию о числе элементов в нем, и при выполнении операции *Union* (x, y) добавлять более короткий список в конец более длинного. Если объединяемые множества содержат примерно поровну элементов, то большого выигрыша не будет, но в целом, как показывает следующая теорема, экономия все-таки имеется.

Теорема 5.4. Предположим, что система непересекающихся множеств реализована с помощью списков вторым способом, и в операции *Union* использована весовая эвристика. Тогда стоимость последовательности из m операций *MakeSet*, *Union* и *FindSet*, среди которых n операций *MakeSet*, есть $O(m+n \cdot log\ n)$ (подразумевается, что первоначально система непересекающихся множеств была пуста).
Доказательство этой теоремы приведено в книге [5].

5.9.4 Реализация с использованием дерева

Пусть каждое множество хранится в виде дерева, и для каждого элемента множества хранится его предок. Представителем множества может являться корень дерева. При таком подходе легко построить пример, когда после нескольких объединений множеств получится ситуация, что множество будет представляться деревом, выродившимся в длинную цепочку, как показано на рис. 5.35. Здесь и далее направление стрелки отображает связь *потомок* → *предок*. В этом случае операция *Union* (x, y) выполняется за $O(1)$, а *FindSet* (x) – за $O(|V|)$.

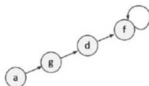


Рис. 5.35. Дерево, выродившееся в цепочку

Применение весовой эвристики при объединении двух множеств, представленных деревьями, оценку для *FindSet* не улучшает. На рис. 5.36 показан пример объединения двух деревьев с учетом весовой характеристики, при этом глубина дерева увеличилась, что нежелательно.

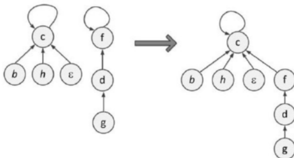


Рис. 5.36. Объединение двух деревьев с учетом весовой характеристики

5.9.5 Эвристика объединением по рангу

В этой реализации для каждой вершины нужно хранить ее высоту (ранг) вершины. Дерево с меньшим рангом подвешивается к дереву с большим рангом, при этом ранг нового дерева не изменяется. Корнем нового дерева становится корень дерева с большим рангом, как на рис. 5.37. Если ранги деревьев равны, то ранг нового дерева увеличивается на единицу.

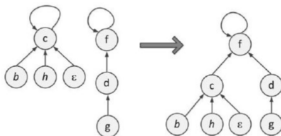


Рис. 5.37. Объединение двух деревьев с учетом рангов

При объединении по рангу оценка времени работы будет примерно такой же, как при списочной реализации с весовой эвристикой.

5.9.6 Эвристика сжатия путей

В данной реализации при определении представителя множества, при прохождении пути от вершины к корню, подвешивается эта вершина и все вершины, составляющие этот путь, к корню. На рис. 5.38 выделена вершина *g*. Будем считать, что от нее найден путь до корня дерева. Тогда подвесим ее и вершину *h* непосредственно к корню. Предполагаемые новые ребра на рисунке показаны пунктирной линией. Выполнив это действие со всеми вершинами, придем к ситуации, когда все они станут сыновьями корня. Заметим, что метод рекурсивный, и перевешивание вершины выполняется на выходе из рекурсии.

Рассмотрим реализацию метода сжатия путей. Пусть элементы множеств – это некоторые целые числа. Их количество равно **`max_n`**. Вся структура данных хранится в виде двух массивов: **`p`** и **`rank`**. Массив **`p`** содержит предков, т. е. **`p[x]`** – это предок элемента **`x`**. Таким образом, мы имеем древовидную структуру данных. Двигаясь по предкам от любого элемента **`x`**, придем к представителю множества, к которому принадлежит **`x`**. Выражение **`p[x] = x`** означает, что *x* является представителем множества, к которому он принадлежит, и, следовательно, корнем дерева.

Массив **`rank`** хранит ранги представителей, его значения имеют смысл только для элементов-представителей. Ранг некоторого элемента-

представителя *x* – это верхняя граница высоты в его дереве. Ранги будут использоваться в операции *Union*.

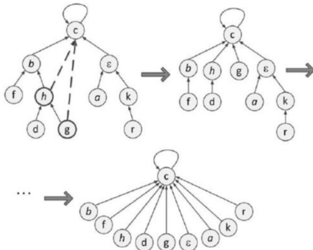


Рис. 5.38. Преобразование дерева

Инициализация СНМ:

```
for (int i=0; i< max_n; ++i)
  p[i] = i;
```

В реализации функции *MakeSet* (*x*) указывается, что элемент *x* является корнем, и его ранг равен нулю.

```
void makeset (int x)
{
  p[x] = x;
  rank[x] = 0;
}
```

В процессе исполнения функции *FindSet*(*x*) осуществляется движение от элемента *x* по предкам, до тех пор, пока не найден представитель. У каждого элемента, через который происходит движение, его значение в **p** исправляется на найденного представителя. Таким образом, фактически реализация функции *FindSet* является двухпроходной: на первом проходе осуществляется поиск представителя, а на втором исправляются значения **p**.

```
int find_set (int x)
{
   if (x == p[x])
      return x;
   return p[x] = find_set (p[x]);
}
```

Рассмотрим реализацию операции *Union* (*x, y*).

```
void union (int x, int y)
{
   x = find_set (x);
   y = find_set (y);
   if (rank[x] > rank[y])
      p[y] = x;
   else
   {
      p[x] = y;
      if (rank[x] == rank[y])
         ++rank[y];
   }
}
```

С помощью функции **find_set** находятся представители множеств, которые содержат элементы **x** и **y**. Затем объединяются два множества в зависимости от следующей ситуации:

- если ранги элементов **x** и **y** различны, то корень с большим рангом становится родительским по отношению к корню с меньшим рангом;
- если же ранги обоих элементов совпадают, родитель выбирается произвольным образом, его ранг увеличивается на 1.

В описанном подходе справедливы следующие оценки быстродействия.

MakeSet (*x*) – выполняется за $O(1)$.

Операции *Union* (*x, y*) и *FindSet* (*x*) выполняются за $O(\alpha(|V|))$, где $\alpha(|V|)$ – обратная функция Аккермана [5], которая растет очень медленно, и для чисел _*int*64 не превосходит 4.

Таким образом, можно считать, что все операции выполняются за константное время.

ГЛАВА 6. ПУТИ В ГРАФЕ

В данном разделе мы перейдем к рассмотрению задач, связанных с нахождением кратчайших путей в графе. Кроме того, рассмотрим проблему существования путей между любыми парами вершин.

Пусть $G = (V, E)$ – ориентированный или неориентированный граф.

Поставим в соответствие каждому ребру $e \in E$ в графе G неотрицательную стоимость $c(e)$, где c: $E \to R^+$ – *функция стоимости*.

Стоимость (*вес*) *пути* p $(v_0, v_1, \dots, v_k)$ определяется как сумма стоимостей ребер, входящих в этот путь: $w(p) = \sum_{i=0}^{k} c(v_{i-1}, v_i)$.

Задача о нахождении кратчайшего пути состоит в нахождении для каждой пары вершин (v, w) пути наименьшей стоимости между ними. Результатом алгоритма поиска кратчайшего пути является последовательность ребер, соединяющая заданные две вершины и имеющая наименьшую суммарную стоимость среди всех таких последовательностей.

Вес кратчайшего пути из u в v равен по определению

$$\delta(u,v) = \begin{cases} \min\{w(p): u \overset{p}{\to} v\}, \text{если существует путь из } u \text{ в } v \\ \infty, \text{иначе} \end{cases}$$

Кратчайший путь из u в v это любой путь из u, для которого $w(p) = \delta(u,v)$.

6.1 Нахождение кратчайшего пути из одного источника

Рассмотрим алгоритм Дейкстры нахождения кратчайших путей [3-5] из одной заданной вершины, которая называется *источником*, во все другие. Будем использовать технику *релаксации* – постепенного уточнения верхней оценки кратчайшего пути в заданную вершину.

Пусть стоимости ребер будут неотрицательными числами. Для каждой вершины v будем хранить величину $d[v]$ – верхнюю оценку кратчайшего пути из вершины источника v_0 в вершину v. Для каждой вершины $v \in V$ мы будем помнить ее предшественника. Как уже было сказано, вес ребра (u,v) есть $c(u,v)$. Релаксация ребра $(u,v) \in E$ состоит в том, что значение $d[v]$ уменьшается до $d[v+c(u,v)]$, если второе значение меньше первого.

Ниже представлена процедура, выполняющая релаксацию ребра (u,v).

процедура *Relax* (u, v, c)
    если $(d[v] > d[u] + c(u,v))$ **то**
        начало
            $d[v] \leftarrow d[u] + c(u,v)$
            отец$[v] \leftarrow u$
        конец
конец процедуры

Важными для алгоритма поиска кратчайшего пути являются следующая лемма и ее следствия.

Лемма 6.1

Отрезки кратчайших путей являются кратчайшими:
если $p(v_1, v_2, \dots, v_k)$ – кратчайший путь из v_1 в v_k и $1 \le i \le j \le k$, то $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$ есть кратчайший путь из v_i в v_j.
Без доказательства

Следствие 6.1.1

Рассмотрим кратчайший путь p из s в v. Пусть (u,v) – последнее ребро этого пути. Тогда $\delta(s, v) = \delta(s, u) + w(u,v)$.

Следствие 6.1.2

Для любого ребра $(u,v) \in E$ справедливо $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Идея алгоритма Дейкстры состоит в построении множества S, содержащего вершины графа, кратчайшие расстояния до которых от источника вычислены: для всех вершин v из этого множества $d[s,v] = \delta(s, v)$. На каждом шаге алгоритма добавляется та из оставшихся вершин, кратчайшее расстояние до которой от источника меньше всех других.

Алгоритм 6.1. Алгоритм Дейкстры

Вход: $G = (V, E)$ – ориентированный граф,
$v_0 \in V$ – источник,
$c: E \to R^+$ – функция стоимости ребер графа G.
Полагаем, что $c(v_i, v_j) = +\infty$, если $(v_i, v_j) \notin E$; $c(v, v) = 0$.

Выход: для всех вершин $v \in V$ наименьшая сумма стоимостей ребер из E, взятая по всем путям, идущим из v_0 в v.

Метод:

начало
  $S \leftarrow \{v_0\}$
  $d[v_0] \leftarrow 0$
  для всех v: $v \in V \setminus \{v_0\}$ **выполнить**
    $d[v] \leftarrow c(v_0, v)$
    отец$[v] \leftarrow$ NULL
  конец цикла
  пока $S \ne V$ **выполнить**
    выбрать вершину $w \in V \setminus S$, для которой $d[w]$ принимает наименьшее значение
    добавить w к S
    для всех v: $v \in$ *смежные*$[w]$ **выполнить**
      Relax (w, v, c)

конец цикла
конец цикла
конец
Конец алгоритма 6.1

Пример

Рассмотрим ориентированный граф на рис. 6.1. В таблице 6.1 показаны шаги работы алгоритма Дейкстры, текущие кратчайшие пути, записанные в массив d, множество S и значения массива *отец.*

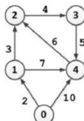


Рис. 6.1. Ориентированный взвешенный граф

Таблица 6.1

Работа алгоритма Дейкстры для графа на рис. 6.1

Шаг	*S*	*w*	*d*[*w*]	*d*					*отец*				
				0	1	2	3	4	0	1	2	3	4
0	{0}	-		0	2	+∞	+∞	10	-	0	0	0	0
1	{0, 1}	1	2	0	2	5	+∞	9	-	0	1	0	1
2	{0, 1, 2}	2	5	0	2	5	9	9	-	0	1	2	1
3	{0, 1, 2, 3}	3	9	0	2	5	9	9	-	0	1	2	1
4	{0, 1, 2, 3, 4}	4	9	0	2	5	9	9	-	0	1	2	1

Алгоритм Дейкстры относится к *жадным* алгоритмам, в которых на каждом шаге делается локально наилучший выбор в надежде, что итоговое решение будет оптимальным. В этом алгоритме на каждом шаге выбирается вершина с наименьшим текущим расстоянием, после чего расстояния до других вершин обновляются. Найденные кратчайшие пути, являются оптимальными. Доказательство этого утверждения можно найти в [3].

Сложность алгоритма Дейкстры зависит от способа нахождения вершины v, а также способа хранения множества вершин и способа обновления расстояний. В простейшем случае, когда для хранения величин d используется массив, время работы алгоритма есть $O(n^2 + m)$, где n – количество вершин, а m – количество ребер. Основной цикл выполняется порядка n раз, в каждом из них на нахождение минимума тратится порядка n операций,

плюс количество перевычислений текущих расстояний, которое не превосходит количества ребер в исходном графе.

6.2 Алгоритм Беллмана - Форда

Алгоритм Беллмана - Форда позволяет находить кратчайшие пути от одной вершины графа до всех остальных, и при этом допускает ребра с отрицательным весом. Он предложен независимо Ричардом Беллманом (*Bellman*) и Лестером Фордом (*Ford*).

Цикл, сумма весов ребер которого отрицательна, называется *отрицательным циклом*.

С помощью алгоритма определяется, существует ли в графе G отрицательный цикл, достижимый из вершины v_0. Для проверки этого факта достаточно выполнить внешнюю итерацию цикла $|V|$ раз. Если при исполнении последней итерации длина кратчайшего пути до какой-либо вершины строго уменьшится, то в графе имеется отрицательный цикл, достижимый из вершины v_0. Следовательно, можно отслеживать изменения в графе и, как только они закончатся, дальнейшие итерации будут бессмысленны.

Алгоритм 6.2. Алгоритм Беллмана - Форда

Вход: $G = (V, E)$ – ориентированный граф,
$v_0 \in V$ – источник, c: $E \to R^+$ – функция стоимости ребер графа G.
Полагаем, что $c(v_i, v_j) = +\infty$, если $(v_i, v_j) \notin E$; $c(v, v) = 0$.

Выход: если в графе нет цикла отрицательного веса, то для всех вершин $v \in V$ наименьшая сумма стоимостей ребер из E, взятая по всем путям, идущим из v_0 в v, иначе 0.

Метод:

начало
  $d[v0] \leftarrow 0$
  для всех v: $v \in V \setminus \{v_0\}$ **выполнить**
    $d[v] \leftarrow c(v_0, v)$
    отец$[v] \leftarrow$ NULL
  конец цикла
  для всех i **от** 1 **до** n **выполнить**
    для всех $(u, v) \in E$ **выполнить**
      Relax (u, v, c)
    конец цикла
  конец цикла
  для всех $(u, v) \in E$ **выполнить**
    если $(d[v] > d[u] + c(u,v)$ **то**
      начало
        выдать 0

выход
конец
конец цикла
для всех v: $v \in V$ **выполнить**
выдать $d[v]$
конец цикла
конец
Конец алгоритма 6.2

Время работы алгоритма Беллмана - Форда оценивается величиной $O(|V| \cdot |E|)$.

6.3 Нахождение кратчайших путей между всеми парами вершин

Алгоритм Флойда - Уоршелла определяет кратчайшие расстояния между всеми вершинами взвешенного графа. Этот алгоритм основан на методе динамического программирования. Он был одновременно опубликован в статьях Роберта Флойда (*Robert Floyd*) и Стивена Уоршелла (*Stephen Warshall*) в 1962 г.

Пусть задан граф $G = (V, E)$, где $V = \{1, 2, 3, \ldots, n\}$. Для реализации алгоритма построим матрицу стоимостей:

$$M[i,j] = \begin{cases} c(i,j), \text{усли дуга}(i,j) \in E \\ +\infty, \text{если дуга } (i,j) \notin E \\ 0, \text{если } i = j \end{cases}$$

В основе алгоритма лежат два свойства кратчайшего пути графа.

1) Отрезки кратчайших путей являются кратчайшими (лемма 6.1).
2) Стоимость кратчайшего пути между вершинами i и j равна сумме стоимостей кратчайших путей между вершинами i и k и вершинами k и j, если вершина k входит в кратчайший путь.

Второе свойство является основой алгоритма. Вычисляется путь p_{ij} от вершины i до вершины j, проходящий через вершины из множества $\{1, 2, \ldots, k\}$.

При $k = 1$ рассматриваются пути, проходящие через вершину 1, при $k = 2$ – через вершины 1, 2, при $k = 3$ – через вершины 1, 2, 3 и т. д.

Пусть на $(k–1)$-м шаге построен кратчайший путь p_{ij} стоимостью d_{ij} с промежуточными вершинами $\{1, 2, \ldots, k–1\}$. При расширении множества промежуточных вершин до k-ой вершины $\{1, 2, ..., k\}$ возможно два исхода.

В первом случае вершина k не входит в кратчайший путь p_{ij}. Тогда от добавления дополнительной вершины ничего не выигрывается, стоимость кратчайшего пути не изменяется.

Во втором случае вершина k входит в кратчайший путь p_{ij}. Кратчайший путь разбивается этой вершиной на два пути: p_{ik} и p_{kj}. Согласно лемме 6.1, p_{ik} и p_{kj} – кратчайшие пути от вершины i до k и от вершины k до j, соответственно. Следовательно, стоимость кратчайшего пути между вершинами i и j равна сумме стоимостей кратчайших путей между вершинами i и k и вершинами k и j.

Пусть d – матрица кратчайших путей между всеми вершинами. Обозначим через $d_{ij}^{(k)}$ стоимость кратчайшего пути из вершины i в вершину j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. Тогда на k-м шаге стоимость кратчайшего пути выражается следующим образом.

$$d_{ij}^{(k)} = \begin{cases} M[i,j], \text{если } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), \text{если } k \geq 1 \end{cases}$$

В результате матрица $d^{(n)}$ будет содержать искомое решение.

Алгоритм Флойда - Уоршелла, формально описанный ниже, реализуется тремя циклами, где внешний цикл с параметром k реализует перебор по количеству промежуточных вершин между всеми парами вершин.

Алгоритм 6.3. Алгоритм Флойда - Уоршелла

Вход: Граф $G = (V, E)$, представленный матрицей стоимостей M, $V = \{1, 2, 3, \ldots, n\}$.

Выход: Матрица кратчайших путей d.

Метод:

начало
  $d \leftarrow M$;
  для всех k **от** 1 **до** n **выполнить**
    для всех i **от** 1 **до** n **выполнить**
      для всех j **от** 1 **до** n **выполнить**
        если $d_{ij} > d_{ik} + d_{kj}$ **то**
          $d_{ij} \leftarrow d_{ik} + d_{kj}$
      конец цикла
    конец цикла
  конец цикла
конец

Конец алгоритма 6.3

Время работы алгоритма определяется тремя вложенными циклами от 1 до n – $O(n^3)$, где n – количество вершин в графе.

Для нахождения последовательности вершин, составляющих кратчайший путь p_{ij}, при изменении d^k_{ij} нужно еще сохранять величину *отец*$^k_{ij}$ – предшественника вершины j на пути от вершины i с множеством промежуточных вершин $\{1 .. k\}$.

Этот алгоритм может определить наличие в графе отрицательных циклов. Если они есть, то после окончания его работы на диагонали матрицы d возникнут отрицательные числа, так как кратчайшее расстояние от вершины в этом цикле до нее самой будет меньше нуля.

Если граф – неориентированный, то все матрицы d^k являются симметричными, поэтому достаточно вычислять элементы, находящиеся только выше (либо только ниже) главной диагонали.

6.4 Транзитивное замыкание графа

Пусть $G = (V, R)$ ориентированный граф. *Транзитивным замыканием* графа G называется граф $G' = (V, R')$, в котором из вершины v в вершину w идет дуга тогда и только тогда, когда существует путь из v в w в графе G.

Формально отношение R' представляется следующим образом:

R': $(a, b) \in R$ & $(b, c) \in R \Rightarrow (a, b) \in R'$ & $(b, c) \in R'$ & $(a, c) \in R'$

Пример

На рис. 6.2. показаны графы, второй из которых является транзитивным замыканием первого.

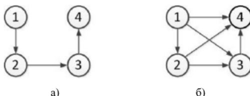


Рис. 6.2. Граф б) – транзитивное замыкание графа а)

Обозначим через $t_{ij}^{(k)}$ наличие пути из вершины i в вершину j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. Пусть M – матрица смежности графа G.

$$t_{ij}^{(k)} = \begin{cases} M[i,j], \text{если } k = 0, \\ t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right), \text{если } k \geq 1. \end{cases}$$

В результате $T^{(n)}$ содержит искомое решение.

Алгоритм 6.4. Построение транзитивного замыкания графа

Вход: Граф $G = (V, E)$, представленный матрицей смежности M, $V = \{1, 2, 3, \ldots, n\}$.

Выход: T – матрица смежности транзитивного замыкания графа G.

Метод:

начало

$T \leftarrow M$

для всех k **от** 1 **до** n **выполнить**

для всех i **от** 1 **до** n **выполнить**

для всех j **от** 1 **до** n **выполнить**

если $t_{ik} = 1$ **и** $t_{kj} = 1$ **то**

$t_{ij} \leftarrow 1$

конец цикла

конец цикла

конец цикла

конец

Конец алгоритма 6.4

Внешний цикл **для всех** с параметром k так же, как и в предыдущем алгоритме, реализует перебор по количеству промежуточных вершин между всеми парами вершин.

Время работы последнего алгоритма и алгоритма Флойда - Уоршелла есть $O(n^3)$, где n – количество вершин в графе.

6.5 Связность в графе

Напомним, что неориентированный граф считается связным, если из любой вершины есть путь в любую другую вершину. Чтобы граф с n вершинами был связным, он должен иметь не менее $(n - 1)$ ребер.

Отношение связности является отношением эквивалентности. Поэтому существует такое разбиение множества вершин графа на классы эквивалентности, что все вершины в каждом подмножестве связаны, а вершины из различных подмножеств не связаны. Каждое такое подмножество вершин графа вместе с ребрами, инцидентными этим вершинам, образует связный подграф. Таким образом, неориентированный граф представим единственным способом в виде объединения непересекающихся связных подграфов.

6.5.1 Компоненты связности

Компонента связности графа – это такое множество вершин графа, что для любых двух вершин из этого множества существует путь из одной в другую, и не существует пути из вершины этого множества в вершину не из этого множества.

Рассмотрим задачу, состоящую в нахождении компонент связности в графе. Для ее выполнения следует провести серию обходов в глубину. Сначала выполняется обход в глубину из любой вершины. Все вершины, которые при этом обходе просмотрены, образуют первую компоненту

связности. Затем находится любая из вершин, не вошедшая в найденную компоненту связности, и выполняется обход в глубину из нее. Таким образом определяется вторая компонента связности. Процесс повторяется до тех пор, пока все вершины не будут просмотрены.

Алгоритм 6.5. Поиск компонент связности

Вход: Граф $G = (V, E)$, все вершины окрашены в белый цвет.

Выход: Граф G, где для всех вершин $v \in V$ определен *номер_КС*[v] – номер компоненты связности, которой вершина v принадлежит.

Метод:

```
начало
    nk ← 0
    для всех вершин u: u ∈ V выполнить
        цвет[u] ← белый
        отец[u] ← NULL
    конец цикла
    для всех вершин u: u ∈ V выполнить
        nk ← nk +1
        если цвет[u] = белый то
            Поиск_КС (u, nk)
    конец цикла
конец

процедура Поиск_КС (u, n)
    цвет [u] ← серый
    номер_КС[u] ← n
    для всех v: v ∈ смежные(u) выполнить
        если цвет[v] = белый то
            начало
                отец [v] ← u
                Поиск_КС (v, n)
            конец
    конец цикла
    цвет[u] ← черный
конец процедуры
```

Конец алгоритма 6.5

Данный алгоритм применим как к неориентированным графам, так и к ориентированным.

Оценка его эффективности – $O\ (n + m)$, так как алгоритм не будет запускаться от одной и той же вершины дважды, а значит, каждое ребро будет просмотрено не более двух раз (ровно два раза для неориентированных графов – с одного конца и с другого конца).

6.5.2 Двусвязность

Пусть $G = (V, E)$ – связный неориентированный граф. Вершину a называют *точкой сочленения* графа G, если существуют такие вершины v и w, что v, w и a различны и всякий путь между v и w содержит вершину a.

Иначе говоря, a – точка сочленения графа G, если удаление вершины a расщепляет G не менее чем на две части. Например, на рис. 6.3 показан граф, в котором вершины с номерами 2, 6, 7 и 8 являются точками сочленения.

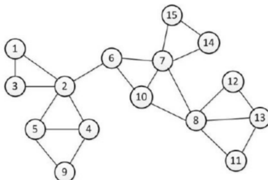


Рис. 6.3. Граф с точками сочленения 2, 6, 7 и 8

Граф G называется *двусвязным*, если для любой тройки различных вершин v, w, a существует путь между v и w, не содержащий a.

Таким образом, неориентированный связный граф двусвязен тогда и только тогда, когда в нем нет точек сочленения. Из определения двусвязности вытекает, что в двусвязном графе между любыми двумя вершинами существует не менее двух различных путей.

На множестве ребер графа G можно задать естественное отношение, полагая, что для ребер e_1 и e_2 выполняется это отношение, если $e_1 = e_2$ или они лежат на некотором цикле.

Легко показать, что это отношение является отношением эквивалентности, разбивающим множество ребер графа G на такие классы эквивалентности $E_1, E_2, \ldots, E_k$, что два различных ребра принадлежат одному и тому же классу тогда и только тогда, когда они лежат на общем цикле.

Для $1 \le i \le k$ обозначим через V_i множество вершин, лежащих на ребрах из E_i. Каждый такой граф $G_i = (V_i, E_i)$ называется *двусвязной компонентой* графа G .

Пример из [3]

На рис. 6.4 показан граф и выделены его двусвязные компоненты:

$E_1 = \{ (v_1, v_2), (v_1, v_3), (v_2, v_3)\}$,

$E_2 = \{ (v_2, v_4), (v_2, v_5), (v_4, v_5)\}$,

$E_3 = \{ (v_4, v_6)\}$,

$E_4 = \{ (v_6, v_7), (v_6, v_8), (v_6, v_9), (v_7, v_9), (v_8, v_9)\}$

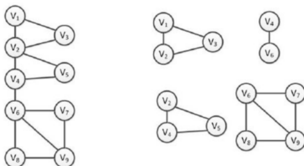


Рис. 6.4. Пример графа и его двусвязных компонент

Лемма 6.2

Пусть $G_i=(V_i, E_i)$ *для* $1 \le i \le k$ – двусвязные компоненты связного неориентированного графа $G = (V, E)$. Тогда

1) граф G_i двусвязен для каждого i, $1 \le i \le k$;
2) для всех $i \ne j$ пересечение $V_i \cap V_j$ содержит не более одной вершины;
3) a – точка сочленения графа G тогда и только тогда, когда $a \in V_i \cap V_j$ для некоторых $i \ne j$.

Без доказательства

Лемма 6.3

Пусть $G = (V, E)$ – связный неориентированный граф, а $S = (V, T)$ – глубинное остовное дерево для него.

Вершина a является точкой сочленения графа G тогда и только тогда, когда выполнено одно из условий:

1) a – корень S и a имеет более одного сына в нем;
2) a – не корень и для некоторого его сына s нет обратных ребер между потомками вершины s (в том числе самой s) и подлинными предками вершины a.

Без доказательства

Пример

Для графа на рис. 6.4 глубинное остовное дерево будет выглядеть, как показано на рис. 6.5.

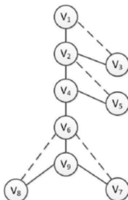


Рис. 6.5. Глубинное остовное дерево для графа на рис. 6.4.

Обратные ребра, не вошедшие в глубинное остовное дерево, обозначены пунктиром. По лемме 6.3 точками сочленения в данном дереве будут вершины v_2, v_4 и v_6. Вершина v_1 не может быть точкой сочленения, так как является корнем глубинного остовного дерева и имеет только одного сына.

6.5.3 Нахождение двусвязных компонент и точек сочленения

Алгоритм поиска двусвязных компонент и точек сочленения основывается на методе поиска в глубину [3]. Во время обхода графа для всех вершин *v* вычисляются числа *пгномер*[*v*]. Они фиксируют последовательность обхода вершин глубинного остовного дерева в прямом порядке.

Кроме того, для каждой вершины *v* вычисляется число

$$\text{нижний}[v] = \min \begin{cases} \textit{пгномер}[v]; \\ \textit{пгномер}[z], \text{где } (v, z) - \text{ обратное ребро;} \\ \textit{нижний}[x], \text{где } x - \text{потомок } v. \end{cases}$$

Точки сочленения определяются следующим образом:

- корень остовного дерева будет точкой сочленения тогда и только тогда, когда он имеет двух и более сыновей;
- вершина *v*, отличная от корня, будет точкой сочленения тогда и только тогда, когда имеет такого сына *w*, что

$$\textit{нижний}[w] \geq \textit{пгномер}\ [v].$$

Алгоритм 6.6. Нахождение двусвязных компонент и точек сочленения

Вход: Связный неориентированный граф $G = (V, E)$.

Выход: Список ребер каждой двусвязной компоненты графа G.

Метод: Пусть T – множество древесных ребер остовного дерева $S = (V, T)$, полученного обходом в глубину графа G.

```
начало
    T ← ∅
    счет ← 1
    для всех вершин u: u ∈ V выполнить
        цвет[u] ← белый
    конец цикла
    выбрать из V произвольную вершину v0
    отец[v0] ← NULL
    Поиск_ДК (v0)
конец

процедура Поиск_ДК (v)
    цвет[v] ← серый
    пгномер[v] ← счет
    счет ← счет + 1
    нижний[v] ← пгномер[v]
    для всех w: w ∈ смежные(v) выполнить
        если цвет[w] = белый то
            начало
                поместить ребро (v, w) в СТЕК
                добавить (v, w) к T
                отец [w] ← v
                Поиск_ДК (w)
                если нижний[w] ≥ пгномер[v] то
                    начало
                        обнаружена двусвязная компонента: вытолкнуть из
                        СТЕКа все ребра вплоть до ребра (v, w)
                    конец
                нижний[v] ← min(нижний[v], нижний[w])
            конец
        иначе
            если (w ≠ отец[v]) то
                начало
                    поместить (v, w) в СТЕК
                    нижний[v] ← min(нижний[v], пгномер[w])
                конец
```

конец цикла
цвет[*v*] ← *черный*
конец процедуры
Конец алгоритма 6.6

Пример

Остовное дерево [3] на рис. 6.5, построенное поиском в глубину, воспроизведено на рис. 6.6, причем вершины переименованы в соответствии со своим значением *пгномер* и для них указаны значения *нижний*.

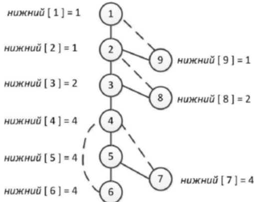


Рис. 6.6. Глубинное остовное дерево с рис. 6.6 со значениями *нижний*

Например, *Поиск_ДК* (6) устанавливает, что *нижний*[6] = 4, так как есть обратное ребро (6, 4). Тогда *Поиск_ДК* (5), который вызвал процедуру *Поиск_ДК* (6), полагает *нижний*[5] = 4, поскольку 4 меньше начального значения *нижний*[5], равного 5.

В результате работы *Поиск_ДК* (5), получается, что *нижний*[5] = 4. Следовательно, вершина 4 – точка сочленения. В этот момент СТЕК содержит следующие ребра (от дна к вершине):

(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 4), (5, 7), (7, 4).

Из СТЕКА выталкиваются все ребра, вплоть до ребра (4, 5). Таким образом, ребра (7, 4), (5, 7), (6, 4), (5, 6) и (4, 5) составляют первую найденную двусвязную компоненту. По окончании работы *Поиск_ДК* (2), обнаруживается, что *нижний*[2] = 1, СТЕК опустошается. Найдена двусвязная компонента, содержащая корень – вершину 1. Так как эта вершина имеет только одного сына, то она не является точкой сочленения.

Теорема 6.1

Алгоритм 6.6 правильно находит двусвязные компоненты графа G с e ребрами и тратит на это время $O(e)$.

Без доказательства

6.6 Построение минимального каркаса графа

Пусть $G(V, E)$ – связный неориентированный граф с заданной функцией стоимости, отображающей ребра в вещественные числа.

Остовное дерево или *каркас* графа – это подграф, который:

- содержит все вершины графа,
- является деревом [5].

Остовное дерево состоит из некоторого подмножества ребер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, и в таком графе нет циклов.

Нас интересуют алгоритмы построения минимального каркаса.

Минимальным каркасом является такой каркас, сумма весов ребер которого минимальна.

Примером задачи, в которой нужно найти минимальный каркас графа, может быть следующая. Имеется n городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой. Известна стоимость строительства каждой дороги между заданными парами городов. Требуется определить, какие дороги нужно построить, чтобы минимизировать общую стоимость строительства.

6.6.1 Алгоритм Краскала

Данный алгоритм построения минимального каркаса был предложен Джозефом Краскалом в 1956 г. Его неформальное описание заключается в следующем.

Будем считать, что каждая вершина графа принадлежит отдельному множеству. Эти множества образуют лес, который в процессе работы алгоритма будет преобразован в одно остовное дерево минимальной стоимости (минимальный каркас). На каждом шаге выбираются ребра в порядке возрастания их весов. Для каждого выбранного ребра выполняются следующие действия:

- если вершины, соединяемые этим ребром, принадлежат разным множествам, то эти множества объединяются в одно, а рассматриваемое ребро добавляется к строящемуся каркасу;
- если вершины, соединяемые этим ребром, принадлежат одному и тому же множеству, то данное ребро исключается из рассмотрения.

Процесс повторяется до тех пор, пока не все множества вершин объединены в одно.

Алгоритм 6.7. Алгоритм Краскала

Вход: Неориентированный граф $G = (V, E)$ с функцией стоимости c, заданной на его ребрах.

Выход: Остовное дерево $S = (V, T)$ наименьшей стоимости для графа G.

Обозначения:

VS – множество непересекающихся множеств вершин, остовный лес;

Q – очередь, содержащая все ребра из E, отсортированные по возрастанию весов.

Метод:

1 $T \leftarrow \varnothing$

2 $VS \leftarrow \varnothing$

3 отсортировать все ребра по стоимости и поместить их в Q

4 **для всех** v: $v \in V$ **выполнить**

5 добавить $\{v\}$ к VS

конец цикла

6 **пока** $|VS| > 1$ **выполнить**

7 выбрать в Q ребро (v, w) наименьшей стоимости

8 удалить (v, w) из Q

9 **если** $v \in W_1$ **и** $w \in W_2$ **и** $W_1 \neq W_2$ **и** $W_1 \in VS$ **и** $W_2 \in VS$ **то**

10 **начало**

11 удалить W_1 *и* W_2 из VS

12 добавить $W_1 \cup W_2$ к VS

13 добавить ребро (v, w) к T

14 **конец**

15 **конец цикла**

Конец алгоритма 6.7

В алгоритме 6.7 в строках 1-2 выполняется инициализация множеств T и VS пустыми множествами. Ребра в E в строке 3 расположены в очереди в неубывающем порядке. В строках 4-5 создается $|V|$ множеств, каждое из которых содержит по одной вершине. В цикле в строках 5-7 выбирается ребро (v, w) с наименьшим весом и удаляется из очереди, а далее в строке 9 определяется, принадлежат ли концы ребра одному и тому же множеству. Если это так, то данное ребро не может быть добавлено к лесу, в противном случае множества, к которым принадлежали концы ребра, объединяются в одно, а ребро добавляется в остовное дерево T.

Алгоритм Краскала является жадным**,** поскольку на каждом шаге ребро минимальной стоимости добавляется к остовному лесу [3]. Время работы алгоритма – $O(|E| \cdot |V|)$.

Пример из [3]

На рис. 6.7 дан неориентированный граф с весами на ребрах. Необходимо для него построить минимальное остовное дерево.

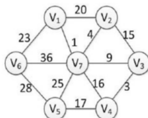


Рис. 6.7. Исходный неориентированный взвешенный граф

Для обозначения множеств введен массив меток вершин графа *Mark*. Начальные значения элементов массива равны номерам соответствующих вершин: *Mark*[*i*] = *i*; $i \in [1..n]$. Ребро добавляется в каркас в том случае, если вершины, соединяемые им, имеют разные значения меток. Изменения *Mark* показаны в таблице 6.2.

Таблица 6.2

Процесс реализации алгоритма для примера на рис. 6.7

Номер итерации	*Ребро – вес*	*Добавление ребра в каркас*	*Значение элементов Mark*
0	–		[1, 2, 3, 4, 5, 6, 7]
1	(v_1, v_7) – 1	+	[1, 2, 3, 4, 5, 6, 1]
2	(v_3, v_4) – 3	+	[1, 2, 3, 3, 5, 6, 1]
3	(v_2, v_7) – 4	+	[1, 1, 3, 3, 5, 6, 1]
4	(v_3, v_7) – 9	+	[1, 1, 1, 1, 5, 6, 1]
5	(v_2, v_3) – 15	–	[1, 1, 1, 1, 5, 6, 1]
6	(v_4, v_7) – 16	–	[1, 1, 1, 1, 5, 6, 1]
7	(v_4, v_5) – 17	+	[1, 1, 1, 1, 1, 6, 1]
8	(v_1, v_2) – 20	–	[1, 1, 1, 1, 1, 6, 1]
9	(v_1, v_6) – 23	+	[1, 1, 1, 1, 1, 1, 1]

На рис. 6.8 представлена последовательность добавления новых ребер в каркас. Получившееся на итерации 9 дерево является каркасом минимального веса.

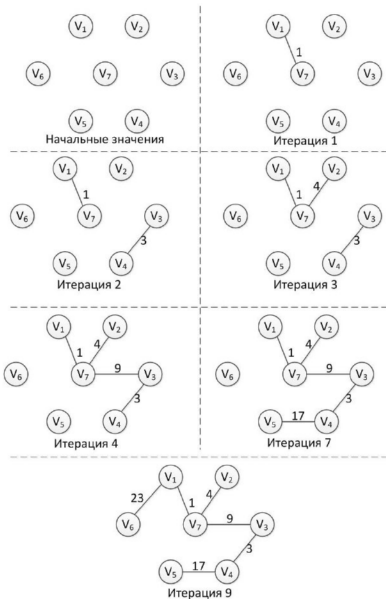


Рис. 6.8. Последовательность добавления новых ребер в каркас

Алгоритм Краскала с СНМ

Использование системы непересекающихся множеств, описанной в разд. 5.9, гарантирует улучшение оценки быстродействия алгоритма Краскала.

Так же, как и в представленной выше версии, сортируются все ребра по неубыванию весов. Для этой цели можно использовать, например, очередь с приоритетами. Каждая вершина помещается в свое дерево с помощью вызова функции *MakeSet*. На это потребуется времени $O(|V|)$.

Затем выбираются ребра, и для каждого ребра определяется, принадлежат ли его концы разным деревьям. Это выполняется с помощью двух вызовов *FindSet* за $O(1)$.

Наконец, объединение двух деревьев будет осуществляться вызовом функции *Union* также за время $O(1)$.

Результирующая оценка работы алгоритма составляет:

$$O(|E| \log |V| + |V| + |E|) = O(|E| \log |V|).$$

6.6.2 Алгоритм Прима

Алгоритм впервые был открыт в 1930 г. чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 г., и, независимо от них в 1959 г., Э. Дейкстрой. Как и в алгоритме Краскала, в алгоритме Прима на каждом шаге к строящемуся остовному дереву добавляется новое ребро наименьшего веса, соединяющее вершины этого остовного дерева с остальными вершинами графа.

Действия алгоритма состоят в следующем. В графе выбирается произвольная вершина – она становится корнем остовного дерева. Далее измеряется расстояние от нее до всех других вершин и выбирается вершина, расстояние до которой минимально. Тем самым находится ребро с наименьшим весом, соединяющее корень дерева с остальными вершинами графа. Эта вершина вместе с ребром добавляются в остовное дерево. Затем пересчитываются расстояния от включенной в дерево вершины до остальных вершин графа. Итак, до тех пор, пока в остовное дерево не добавлены все вершины графа.

При реализации алгоритма надо уметь на каждом шаге быстро пересчитывать расстояния от невключенных в остовное дерево вершин до остовного дерева. Для этого можно воспользоваться очередью с приоритетами Q, в которой будут находиться вершины графа, еще не попавшие в остовное дерево. Приоритет вершины v определяется значением *приор*[v], которое равно минимальному весу ребер, соединяющих вершину v с вершинами минимального остовного дерева. Поле *отец*[v] для вершин дерева указывает на родителя.

Алгоритм 6.8. Алгоритм Прима

Вход: Граф $G = (V, E)$ и вершина $r \in V$ – корень минимального остовного дерева, функция стоимости c: $E \rightarrow R^+$.

Выход: Остовное дерево $S = (V, T)$ наименьшей стоимости для графа G.

Метод:

```
начало
    Q ←V
    T ←∅
    для всех u: u ∈ Q выполнить
        приор[u] ← ∞
    конец цикла
    приор[r] ← 0
    отец[r] = NULL
    пока Q ≠ ∅ выполнить
        выбрать в Q вершину u с минимальным значением приор[u]
        удалить u из Q
        ребро (отец[u], u) добавить к T
        для всех v: v ∈ смежные(u) выполнить
            если v ∈ Q & c(u, v) < приор[v] то
            начало
                отец[v] ← u
                приор[v] ← c(u, v)
            конец
        конец цикла
    конец цикла
конец
```

Конец алгоритма 6.8

На рис. 6.9 показана схема работы алгоритма Прима с корнем в вершине v_1.

Введем массив текущих расстояний от построенной части остовного дерева до остальных вершин графа. Для примера, рассмотренного выше, в таблице 6.3 показан результат действия алгоритма, а именно: итерации, выбранные вершины и ребра с их весовыми значениями.

Время работы алгоритма без использования очереди с приоритетами – $O(|V|\cdot|E|)$, с использованием же очереди – $O(|E|\cdot\log|V|)$. Существуют различные реализации, например, с использованием фибоначчиевой кучи, в которых это время улучшено, но их рассмотрение выходит за рамки данного учебного пособия.

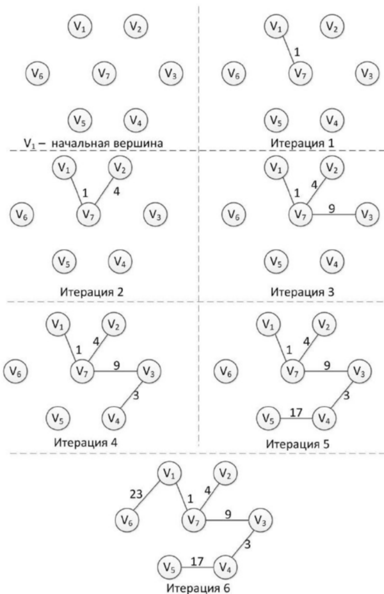


Рис. 6.9. Последовательность добавления новых ребер в остовное дерево алгоритмом Прима

Таблица 6.3

Результат работы алгоритма 6.8 для примера графа на рис. 6.9

Номер итерации	*Массив текущих расстояний*								*Вершина, ребро – вес*
0	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	$V_1 – 0$
	Откуда								
	Вес	0	∞	∞	∞	∞	∞	∞	
1	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_7 , (V_1, V_7) – 1
	Откуда		V_1				V_1	V_1	
	Вес	0	20	∞	∞	∞	23	1	
2	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_2 , (V_2, V_7) – 4
	Откуда		V_7	V_7	V_7	V_7	V_1	V_1	
	Вес	0	4	9	16	25	23	1	
3	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_3 , (V_3, V_7) – 9
	Откуда		V_7	V_7	V_7	V_7	V_1	V_1	
	Вес	0	4	9	16	25	23	1	
4	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_4 , (V_3, V_4) – 3
	Откуда		V_7	V_7	V_3	V_7	V_1	V_1	
	Вес	0	4	9	3	25	23	1	
5	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_5, (V_4, V_5) – 17
	Откуда		V_7	V_7	V_3	V_4	V_1	V_1	
	Вес	0	4	9	3	17	23	1	
6	Куда	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_6 , (V_1, V_6) – 23
	Откуда		V_7	V_7	V_3	V_4	V_1	V_1	
	Вес	0	4	9	3	17	23	1	

6.6.3 Алгоритм Прима с удалением ребер

При работе данного алгоритма на каждом шаге из графа вычеркивается ребро максимальной стоимости с тем условием, что удаление этого ребра не разрывает граф на две или более компоненты связности. Таким образом, после удаления ребра граф должен оставаться связным.

Для того чтобы определить, остался ли граф связным, достаточно запустить поиск в глубину от одной из вершин, связанных с удаленным ребром. Если в результате другая вершина из этого ребра окажется достижимой, то ребро удалить можно.

Условием продолжения работы алгоритма может служить, например, тот факт, что количество ребер в графе больше либо равно количеству вершин. Как только количество ребер станет на единицу меньше, чем количество вершин, нужно остановиться.

Алгоритм 6.9. Алгоритм Прима с удалением ребер

Вход: Граф $G = (V, E)$ и функция стоимости $c: E \to R^+$.

Выход: Остовное дерево $S = (V, T)$ наименьшей стоимости для графа G.

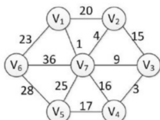
Метод:

```
начало
    T ← E
    пока |T| ≠ |V|–1 выполнить
        выбрать в T ребро максимального веса (u, v)
        удалить ребро (u, v) из T
        если нет_пути (u, v) то
            добавить ребро (u, v) к T
    конец цикла
конец

процедура Поиск (u)
    цвет [ u ] ← серый
    для всех v: v ∈ смежные(u) выполнить
        если цвет[v] = белый то
            Поиск(v)
    конец цикла
    цвет[u] ← черный
конец процедуры

функция нет_пути (x, y) : логический
    для всех вершин u: u ∈ V выполнить
        цвет[u] ← белый
    конец цикла
    Поиск(x)
    если цвет[y] = белый то
        выдать истина
    иначе
        выдать ложь
конец функции
```

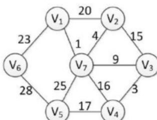
Конец алгоритма 6.9

Для графа из примера на рис. 6.9 последовательность построения минимального каркаса будет выглядеть, как показано на рис. 6.10.

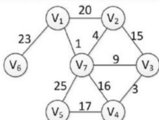
Поскольку на каждом шаге запускается метод поиска в глубину, время работы этого алгоритма существенно хуже, чем в предыдущем алгоритме и оценивается как $O(|E|\cdot(|V|+|E|))$.



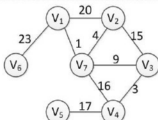
Исходный граф



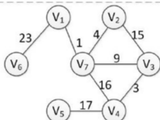
Удалили ребро (v_6, v_7)



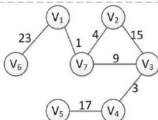
Удалили ребро (v_6, v_5)



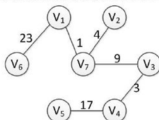
Удалили ребро (v_5, v_7)



Удалили ребро (v_1, v_2)



Удалили ребро (v_4, v_7)



Удалили ребро (v_2, v_3)

Рис. 6.10. Последовательность удаления ребер из графа алгоритмом 6.9 для получения минимального каркаса

6.7 Циклы в графе

Пусть дан ориентированный или неориентированный граф без петель и кратных ребер. При решении различных задач с использованием такого графа часто требуется проверить, является ли он ациклическим, и если не является, то найти какой-нибудь цикл. Для решения подобной задачи можно воспользоваться методом поиска в глубину, и по массиву предков восстановить цикл.

В этом разделе рассматриваются циклы особого вида. Предполагается, что граф может содержать кратные ребра.

6.7.1 Эйлеровы циклы

Если граф имеет цикл, не обязательно простой, содержащий все ребра графа по одному разу, то такой цикл называется *эйлеровым циклом*, а граф называется *эйлеровым графом*.

Эйлеров цикл содержит не только все ребра, но и все вершины графа, причем, возможно, не по одному разу. Ясно, что эйлеровым может быть только связный граф. На рис. 6.11 приведены примеры эйлеровых графов.

Эйлеров граф можно нарисовать на бумаге, не отрывая от нее карандаша, не проходя при этом уже по обведенному ребру вторично. Вышеопределенные понятия распространяются аналогично на мультиграфы.

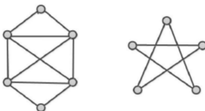


Рис. 6.11 Примеры эйлеровых графов

Леонард Эйлер (1707 – 1783 гг.) первым в своей знаменитой задаче о Кенигсбергских мостах (1736 г.) рассмотрел вопрос о существовании таких циклов в графах [7]. Кенигсберг (теперь Калининград) расположен на обоих берегах реки Преголя и на двух островах этой реки. Берега реки и два острова были соединены в то время семью мостами, как показано на карте (рис. 6.12).

Можно ли, начав с некоторой точки, совершить прогулку и вернуться в исходную точку, пройдя по каждому мосту ровно один раз? Для ответа построим граф (рис. 6.12), в котором каждому участку суши поставлена в соответствие вершина. Две вершины соединяются ребром тогда и только тогда, когда соответствующие участки суши соединены мостом. Требуется определить, будет ли построенный граф эйлеровым.

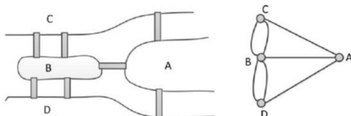


Рис. 6.12 Карта города и соответствующий ей граф

Необходимые и достаточные условия существования эйлерова цикла сформулированы в следующей теореме.

Теорема 6.2

Если неориентированный граф G связен и в нем более одной вершины, то следующие утверждения эквивалентны:

- G – эйлеров граф.
- Каждая вершина G имеет четную степень.
- Множество ребер G можно разбить на простые циклы.

Доказательство

Докажем, что из утверждения 1 следует утверждение 2. Поскольку G – эйлеров граф, то эйлеров цикл, проходя через каждую вершину, входит в нее по одному ребру, выходит по другому. Следовательно, каждой вершине инцидентно четное число ребер. А поскольку цикл содержит все ребра, то отсюда следует четность степеней всех вершин.

Докажем обратное, т. е. что из утверждения 2 следует утверждение 1. Итак, все степени вершин графа четны. Доказательство будет состоять из построения цикла, содержащего все ребра графа G. Начнем строить цикл P_1 из произвольной вершины v_1 и будем продолжать его насколько это возможно, выбирая каждый раз новое ребро. Так как степени вершин четны, то попав в очередную отличную от v_1 вершину, будем иметь в распоряжении еще непройденное ребро. Добавим его в цикл P_1.

Построение цикла закончится в вершине v_1. Если P_1 содержит все ребра, то построение закончено. Иначе удалим из графа все ребра, принадлежащие P_1 .

Рассмотрим граф G_1, полученный в результате удаления этих ребер. G и P_1 имели вершины только четных степеней, следовательно, граф G_1 будет также иметь вершины только четных степеней. В силу связности P_1 и G_1 будут иметь хотя бы одну общую вершину v_2.

Начиная с вершины v_2, построим цикл P_2 в G_1. Пусть P_1' и P_1'' – части цикла P_1 от вершины v_1 до вершины v_2 и от вершины v_2 до вершины v_1, соответственно. Новый цикл $P_3 = P_1' \cup P_2 \cup P_1''$ начинается в v_1, проходит по

всем ребрам P_1' до v_2, затем по всем ребрам P_2 и возвращается в v_1 по ребрам из P_1''.

Процесс построения продолжается до тех пор, пока не получим эйлеров цикл.

Доказательство эквивалентности утверждений 2 и 3 авторы оставляют для самостоятельного изучения.

Пример

Рассмотрим граф G (см. рис. 6.13), в качестве начальной вершины выберем вершину 2.

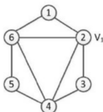


Рис. 6.13 Граф G

Построим цикл P_1, как показано на рис. 6.14, начав с вершины 2 и заканчивая в ней же. В качестве общей вершины, принадлежащей графу G и циклу P_1, выберем, например, вершину 6. Обозначим часть цикла от вершины 6 до вершины 2 за P_1', а от вершины 2 до вершины 6 за P_1''.

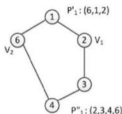


Рис. 6.14 Цикл P_1 в графе G

Начиная с вершины 6, построим цикл P_2 (рис. 6.15).

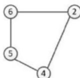


Рис. 6.15 Цикл P_2 в графе G

Получим новый цикл: $P_3 = P_1' \cup P_2 \cup P_1''$, который начинается в вершине 2, проходит по всем ребрам P_1' до вершины 6, затем все ребра P_2 и возвращается в вершину 2 по ребрам из P_1''.

6.7.2 Алгоритмы поиска эйлерова цикла

В данном разделе описывается два алгоритма построения эйлерова цикла. Оба алгоритма на вход получают эйлеровы графы.

Алгоритм Флери

В данном алгоритме нужно указать очередность обхода ребер в цикле, присвоив им соответствующие порядковые номера 1, 2, …, $|E|$, так, чтобы номер, присвоенный ребру, указывал, каким по счету это ребро проходится в эйлеровом цикле. В процессе обхода графа пройденные ребра удаляются. Алгоритм заканчивается, когда все ребра вычеркнуты.

Введем следующее определение.

> *Мост* – это ребро, удаление которого из графа приводит к тому, что граф распадается на несколько компонент связности.

Алгоритм 6.10. Алгоритм Флери

Вход: Эйлеров граф $G(V, E)$, заданный списками смежностей.
Выход: Последовательность ребер эйлерова цикла.
Метод:

Шаг 0:
  $k \leftarrow 1$
  выбрать произвольную вершину $u \in V$
  присвоить произвольному ребру $(u,v) \in E$ номер k
  удалить ребро (u, v) из E
  перейти в вершину v

Шаг 1: выбрать любое ребро $(v, w) \in E$, причем *мост* выбрать только в том случае, если нет других возможностей
  $k \leftarrow k + 1$
  присвоить ребру (v, w) номер k
  удалить ребро (v, w) из E
  перейти в вершину w
  $v \leftarrow w$

Шаг 2:
  если $k < |E|$ **то**
    перейти на Шаг 1
  иначе
    выход

Конец алгоритма 6.10

Пример

На рис. 6.16 показан граф, в котором ребра пронумерованы в порядке обхода алгоритмом Флери.

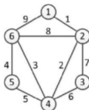


Рис. 6.16. Нумерация ребер в порядке обхода эйлерова цикла алгоритмом Флери

Заметим, что этот алгоритм является модифицированным поиском в глубину. Поэтому из того, что время поиска в глубину есть $O(|V|+|E|)$, и для каждого ребра надо проверить, является ли оно мостом, сложность алгоритма есть $O\ (n{\cdot}(n{+}m))$, где $n = |V|,\ m = |E|$.

Линейный алгоритм поиска эйлерового цикла

Этот алгоритм также похож на алгоритм поиска в глубину. Он состоит в построении пути, который начинается с произвольной вершины v. На каждом шаге для дальнейшего продвижения выбирается еще непройденное ребро. При этом пройденные ребра из графа удаляются, а вершины, входящие в строящийся путь, запоминаются в стеке. Процесс продолжается до тех пор, пока множество смежных вершин очередной вершины u не окажется пустым. Так как степени всех вершин графа четны, в этот момент вершина u совпадет с начальной вершиной v, а пройденные ребра образуют цикл, но он может включать не все ребра графа. Вершина v выводится в качестве первой вершины эйлерового цикла.

Процесс продолжается, начиная с вершины, стоящей на верхушке стека. Для обнаружения еще непройденных ребер необходимо пройти в обратную сторону по построенному пути, который хранится в стеке, до тех пор, пока не встретится вершина v, которой инцидентно непройденное ребро $(v,\ u)$. Все вершины пути до этой вершины u из стека выталкиваются и записываются в эйлеров цикл. Так как граф связен, такая вершина обязательно встретится. Процесс заканчивается, когда обнаруживается, что стек пуст.

Алгоритм 6.11. Поиск эйлерова цикла с использованием стека

Вход: Эйлеров граф $G\ (V,\ E)$, заданный списками смежностей.
Выход: Последовательность вершин эйлерового цикла.

Обозначения:

смежные (v) – множество вершин, смежных с вершиной v,

S – стек для хранения вершин.

Метод:

```
начало
    S ← ∅
    выбрать произвольную вершину v ∈ V
    push(S, v)
    пока S ≠ ∅ выполнить
        v ← top(S)
        если смежные(v) = ∅ то
            начало
                v ← pop(S)
                вывод v
            конец
        иначе
            начало
                выбрать произвольную вершину u ∈ смежные(v)
                push(S, u)
                смежные(v) ← смежные(v) \ {u}
                смежные(u) ← смежные (u) \ {v}
            конец
    конец цикла
конец
```

Конец алгоритма 6.11

Для обоснования алгоритма сначала заметим, что первой в стек помещается начальная вершина v, и из стека она будет удалена последней.

Покажем, что в конечном итоге каждое ребро будет пройдено. Действительно, допустим, что в момент окончания работы алгоритма имеются еще непройденные ребра. Поскольку граф связен, должно существовать хотя бы одно непройденное ребро, инцидентное посещенной вершине. Но тогда эта вершина не могла быть удалена из стека, и стек не мог стать пустым.

Будем говорить, что ребро (x,y) представлено в стеке, если в какой-то момент работы алгоритма в стеке рядом находятся вершины x и y. Ясно, что каждые две вершины, расположенные рядом в этом стеке, образуют ребро. Допустим, в какой-то момент из стека удаляется вершина x, а непосредственно под ней в стеке находится вершина y. Возможно, что вершина y будет перемещена из стека при следующем повторении цикла, тогда ребро (x,y) будет пройдено. Если между удалением из стека вершины x и следующей за ней вершины будет пройдена некоторая последовательность ребер, начинающаяся в вершине y, то в виду четности степеней эта

последовательность может закончиться только в вершине y. Значит, и в этом случае следующей за вершиной x из стека будет удалена вершина y. Из этого рассуждения видно, что последовательность выводимых вершин образует путь, и что каждое ребро графа в конечном итоге будет содержаться в этом пути, причем один раз.

При каждой итерации цикла в рассмотренном алгоритме либо проходится одно ребро, либо одна вершина удаляется из стека. Общая трудоемкость этого алгоритма оценивается как $O(n + m)$.

Заметим, что способ хранения графа должен обеспечить возможность быстрого просмотра множества ребер, инцидентных данной вершине. Подходящим является, например, задание графа списками смежности.

Рекурсивная реализация поиска эйлерового цикла представлена в следующей схеме.

процедура *cycle_search*(u)
  для всех непройденных ребер (u, v) **выполнить**
    (u, v) – отметить и удалить из списка смежности
    cycle_search(v)
  конец цикла
  печать (u)
конец процедуры

Пример. Задача о рыбе

Дан набор костяшек домино, каждая пронумерована с левой и правой стороны. Они поочередно ставятся одна за другой в соответствии с номерами, указанными на сторонах, равными значениями друг к другу. Нужно определить, можно ли из них построить рыбу: так расположить костяшки, чтобы они были все использованы, и последовательность начиналась и заканчивалась одним и тем же числом.

Решение этой задачи сводится к нахождению эйлерова цикла. Для этого каждую кость домино представим ребром графа.

Рассмотрим следующий набор костяшек: (1, 2), (1, 3), (1, 4), (1, 5), (2, 4), (2, 5), (2, 0), (0, 3), (0, 4), (0, 6), (6, 3), (6, 4), (6, 5), (5, 4), (3, 4).

Граф, соответствующий этому набору показан на рис. 6.17.

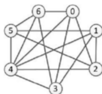


Рис. 6.17. Граф для примера задачи о рыбе

Так как степень каждой вершины этого графа четна, то граф – эйлеров, и мы можем построить эйлеров цикл, начиная из любой его вершины.

6.7.3 Гамильтоновы циклы

Гамильтоновым циклом (*путем*) называют простой цикл (путь), содержащий все вершины графа.

Граф, содержащий гамильтонов цикл, называется *гамильтоновым графом*.

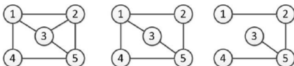


Рис. 6.18. Примеры графов

Пример

Среди трех графов на рис. 6.18 только первый слева граф – гамильтонов.

Теорема 6.3 *(Дирак,* 1952 г.) Если в простом графе с $n \geq 3$ вершинами для любой вершины степень $p(v) \geq n / 2$, то граф является гамильтоновым.
Без доказательства

Внешне определение гамильтонова цикла похоже на определение эйлерова цикла. Однако есть кардинальное различие в сложности решения задачи распознавания и построения. Для эйлерова цикла имеется эффективный алгоритм его построения. Для гамильтоновых же циклов все известные алгоритмы требуют перебора большого числа вариантов.

Рассмотрим алгоритм построения гамильтонова цикла, использующий поиск в глубину, но не в самом графе, а в дереве путей. Вершинами этого дерева являются всевозможные простые пути, начинающиеся в некоторой вершине v (рис. 6.19). Действия алгоритма состоят в рассмотрении этих путей. Путь просматривается до тех пор, пока не будет обнаружен гамильтонов цикл или пока все возможные пути не будут исследованы.

На каждом шаге алгоритма имеется уже построенный отрезок пути, он хранится в стеке *PATH*. Для каждой вершины x, входящей в *PATH*, хранится множество $N(x)$ всех вершин, смежных с вершиной x, которые еще не рассматривались в качестве возможных продолжений пути из вершины x. Когда вершина x добавляется к пути, множество $N(x)$ полагается равным множеству всех смежных вершин. В дальнейшем рассмотренные вершины удаляются из этого множества.

Далее происходит проверка того, что если $N(x) \neq 0$ и в $N(x)$ имеются вершины, не помещенные в стек, то одна из таких вершин добавляется в стек. В противном случае вершина x исключается из стека. Когда после добавления в стек очередной вершины оказывается, что путь содержит все

вершины графа, остается проверить, смежны ли первая и последняя вершины пути, записанного в стек, и при утвердительном ответе выдать очередной гамильтонов цикл.

Алгоритм 6.12. Поиск гамильтоновых циклов

Вход: Неориентированный связный граф $G = (V, E)$

Выход: Последовательности вершин, образующие все гамильтоновы циклы в графе

Метод:

```
начало
    выбрать произвольную вершину v ∈ V
    push (PATH, v);
    N(v) ← смежные (v) ;
    пока PATH ≠∅ выполнить
        x ← top(PATH)
        если N(x) ≠∅ то
            начало
                взять произвольную вершину y ∈ N(x)
                N(x) ← N(x) \ {y}
                если вершина y не находится в PATH то
                    начало
                        push (PATH, y)
                        N(y) ← смежные (y);
                        если PATH содержит все вершины то
                            если y ∈ смежные (v) то
                                выдать цикл из PATH
                    конец
            конец
        иначе
            удалить вершину x из PATH
    конец цикла
конец
```

Конец алгоритма 6.12

На рис 6.19 показаны гамильтонов граф и соответствующее ему дерево путей из вершины 1.

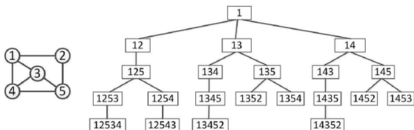


Рис. 6.19. Гамильтонов граф и дерево путей для него

В худшем случае время работы описанного алгоритма растет с факториальной скоростью. Однако можно улучшить время поиска, если при обходе дерева путей проверять на связность каждый граф, получающийся из исходного удаления всех вершин рассматриваемого пути. Данная проблема остается за пределами учебного пособия.

Ниже показан пример реализации алгоритма с помощью рекурсии, где ***k*** – это номер итерации, ***path*** – массив для хранения вершин, составляющих строящийся путь, ***new*** – массив признаков: просмотрена вершина или нет, ***A*** – матрица смежности графа размера $n \times n$.

```
#define N 100
int path[N] = {0};
int new[N] = {0};
int A[N][N];
int n;

void Gm( int k )
{
   int i, j, v;
   v = path[k - 1];               // номер последней вершины
   for (j = 0; j < n; j++)
      if (A[v][j] != 0)           // есть ребро (v,j)
         if((k == n)&&(j == 0))
            Print(path);          // вывод цикла
         else
            if ( new[j] )
            {
               path[ k ] = j;
               new[ j ] = 0;
               Gm( k + 1 );
               new[ j ] = 1;      // поиск другого цикла
            }
}
```

В основной программе следует вызвать описанную функцию **Gm(1)**.

ГЛАВА 7. ПОТОКИ В СЕТЯХ

Материал данной главы основан на содержимом главы «Задача о максимальном потоке» классической книги [5].

Рассмотрим сеть труб, по которым некоторое вещество движется от источника к единому стоку. Эту сеть можно представить ориентированным графом. Пометки ребер в нем будут обозначать пропускную способность труб. Такой же граф можно использовать при решении задач, например, о движении тока по проводам, о передаче информации по линиям связи, о перевозке товаров от производителя к потребителю и т. д. В последнем случае пометки ребер, например, будут представлять ширину дорог. В таких задачах обычно требуется вычислить *максимальный поток* в данной сети.

Сетью называется ориентированный граф $G = (V, E)$, каждому ребру $(u, v) \in E$ которого поставлено в соответствие число $c(u, v) > 0$, называемое *пропускной способностью* ребра.

В случае, когда $(u, v) \notin E$, $c(u, v) = 0$.

В сети выделяются две вершины: *исток s* и *сток t*. Для удобства полагается, что любая вершина лежит на некотором пути из истока к стоку. Таким образом, граф G является связным, и $|E| \geq |V| - 1$.

На рис. 7.1 показан пример сети. На ребрах указаны их пропускные способности.

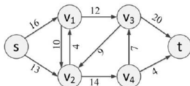


Рис. 7.1. Пример сети

Пусть дана сеть $G = (V, E)$, пропускная способность которой задается функцией c.

Потоком в сети G назовем функцию $f : V \times V \to R$, удовлетворяющую трем свойствам:

1) Ограничение пропускной способности:

$$f(u, v) \leq c(u, v) \text{ для всех } u, v \in V.$$

2) Антисимметричность: $f(u, v) = -f(v, u)$ для всех u, v из V.

3) Сохранение потока: $\sum_{v \in V} f(u, v) = 0$ для всех $u \in V \setminus \{s, t\}$.

Величина потока f определяется как $|f| = \sum_{v \in V} f(s, v)$.

Таким образом, при определении величины потока суммируются потоки по всем ребрам, выходящим из истока.

Задача о максимальном потоке состоит в следующем: для заданной сети G с истоком s и стоком t найти поток максимальной величины.

На рис 7.2 показан поток величины 19, на ребрах сети первое число указывает величину потока, второе – пропускную способность ребра, если же $f(u, v) = 0$, то поток не указывается.

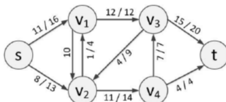


Рис. 7.2. Сеть с указанием потока по каждому ребру

Рассмотрим свойства потока. Из первого следует, что поток из одной вершины в другую не превышает пропускной способности ребра, соединяющего эти вершины. Из второго свойства вытекает, что величина $f(u,v)$ может быть как положительной, так и отрицательной, где отрицательные значения соответствуют движению в обратную сторону. Тогда $f(u, u) = 0$ для любой вершины u. Из свойства сохранения потока следует, что для любой вершины u, кроме стока и истока, сумма потоков во все другие вершины равна нулю. Поэтому, если вершина не является истоком или стоком, в ней вещество не накапливается.

Учитывая антисимметричность, свойство 3 можно переписать иначе:

$$\sum_{u \in V} f(u,v) = 0,$$

т. е. сумма всех потоков из других вершин в вершину v равна нулю.

Если вершины u и v не соединены ребром, то поток между ними равен нулю. Это следует из того, что если $(u, v) \notin E$ и $(v, u) \notin E$, то $c(u, v) = c(v, u) = 0$. Тогда из первого свойства следует, что $f(u, v) \le 0$ и $f(v, u) \le 0$. Вспоминая, что $f(u, v) = -f(v, u)$, получим: $f(u, v) = f(v, u) = 0$.

Разделим вещество, поступающее в данную вершину v и вещество, из нее выходящее, то есть положительные и отрицательные значения $f(u,v)$.

Суммарный положительный поток, *входящий* в вершину задается выражением $\sum_{\substack{u \in V \\ f(u,v)>0}} f(u,v)$.

Аналогично определяется суммарный положительный поток, *выходящий* из некоторой вершины.

Суммарный чистый поток в некоторой вершине равен разности суммарного положительного потока, выходящего из данной вершины, и суммарного положительного потока, входящего в нее.

Тогда свойство сохранения потока формулируется так: для любой вершины, кроме истока и стока, входящий в нее суммарный положительный поток равен выходящему суммарному положительному потоку.

Пример

Рассмотрим пример из [5]. Компания производит изделия на фабрике в городе A (исток s) и складирует их в городе B (сток t). Она арендует место в грузовиках другой фирмы, и место это ограничено: из города u в город v можно доставить не более $c(u,v)$ ящиков в день. Ограничения $c(u, v)$ показаны на рис.7.1, на каждом ребре написано максимальное число ящиков, которые можно отправить в день.

Задача состоит в том, чтобы ежедневно перевозить максимально возможное количество изделий из города A в город B. При этом путь может занимать несколько дней, и ящики могут ждать отправки в промежуточных пунктах, но необходимо, чтобы для каждого пункта число ежедневно прибывающих ящиков было равно числу увозимых, иначе ящиков не хватит или они будут накапливаться. Величиной потока будет число изделий, ежедневно отгружаемых из города A. Нас интересует поток максимальной величины.

Заметим, что модель не учитывает встречные перевозки. Если из пункта v_1 в v_2 ежедневно везут восемь ящиков, а из v_2 в v_1 ежедневно везут три ящика, чему должны быть равны $f(v_1,v_2)$ и $f(v_2,v_1)$? Так как эти величины должны быть противоположны, положим $f(v_1,v_2) = 8 - 3 = 5$, а $f(v_2,v_1)= -5$. Те же значения функции f соответствуют ежедневным перевозкам пяти ящиков из пункта v_1 в v_2, так что в модели встречные перевозки автоматически сокращаются.

7.1 Сокращение потоков между вершинами

В сети на рис.7.1 пропускная способность ребра (v_1, v_2,) равна 10, а ребра (v_2, v_1) – 4, что отображено на рис. 7.3 *a*. Пусть сначала из пункта $v1$ в пункт $v2$ возили ежедневно 8 ящиков (рис. 7.3 *b*). Затем стали возить 3 ящика в обратном направлении (рис. 7.3 *c*). Потом уменьшили число перевозимых в обратную сторону ящиков на 3 (рис. 7.3 *d*). Эти две различные ситуации соответствуют одной и той же функции f, а именно, в обоих случаях $f(v_1,v_2) = 5$, а $f(v_2, v_1) = -5$.

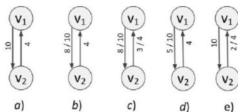


Рис. 7.3. Сокращение потоков между вершинами v_1 и v_2 в сети с рис. 7.2

Если требуется перевозить дополнительно в день 7 ящиков из пункта v_2 в v_1, то нужно, прежде всего, отменить перевозку 5 ящиков в обратную сторону, после чего назначить перевозку дополнительных 2 ящиков, что и отображено на рис. 7.3 *e*.

7.2 Несколько истоков

Задача о максимальном потоке для нескольких истоков и стоков сводится к рассмотренной выше построением эквивалентной сети с одним истоком и одним стоком. А именно: в сеть добавляется *общий* исток *s,* из которого ведут ребра с бесконечной пропускной способностью во все прежние истоки. Аналогичным образом из всех прежних стоков проведены ребра в *общий* сток *t.*

В примере из [5] на рис. 7.4 легко видеть, что каждый поток в сети (а) соответствует потоку в сети (б) и наоборот.

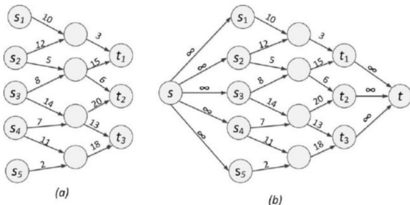


Рис. 7.4. Преобразование сети с несколькими истоками и стоками (*a*) к сети с одним истоком и с одним стоком (*b*)

7.3 Остаточная сеть

Будем использовать следующее соглашение: если в выражении на месте вершины стоит множество вершин, то имеется в виду сумма по всем элементам этого множества (*неявное суммирование*). Это относится и к случаю нескольких переменных. Например, если X и Y – множества вершин, то

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$$

.

В этих обозначениях закон сохранения потока запишется как $f(u,V) = 0$ для всех $u \in V \setminus \{s, t\}$. Кроме того, в неявных суммах мы опускаем фигурные скобки.

Лемма 7.1

Пусть f – поток в сети $G = (V, E)$. Тогда

- для любого $X \subseteq V$ выполнено: $f(X, X) = 0$;
- для любых $X, Y \subseteq V$ выполнено: $f(X, Y) = -f(Y, X)$;
- для любых $X, Y, Z \subseteq V$ из $X \cap Y = \varnothing$ следует:

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z) \text{ и } f(Z, X \cup Y) = f(Z, X) + f(Z, Y).$$

Без доказательства

Пусть дана сеть $G = (V, E)$ с истоком s и стоком t, f – поток в этой сети. Для любой пары вершин u и v *остаточная пропускная способность* из вершины u в вершину v определяется как:

$c_f(u, v) = c(u, v) - f(u, v)$.

Остаточная пропускная способность показывает, сколько еще потока можно направить из u в v. Например, если $c(u,v) = 16$, а $f(u,v) = 11$, то по ребру (u, v) можно переслать еще 5 единиц.

Заметим, что остаточная пропускная способность $c_f(u, v)$ может превосходить $c(u, v)$, если в данный момент поток $f(u, v)$ отрицателен. Например, если $c(u, v) = 16$, а $f(u, v) = -4$, то $c_f(u,v) = 20$. Таким образом, мы можем увеличить поток на 4, отменив встречный поток, и еще отправить 16 единиц, не превышая пропускной способности ребра (u, v).

Неформально говоря, остаточная сеть состоит из тех ребер, поток по которым можно увеличить.

Сеть $G_f = (V, E_f)$, где $E_f = \{ (u, v) \in V{\times}V\colon c_f(u, v) > 0 \}$, называется *остаточной сетью* сети G, порожденной потоком f. Ее ребра, называемые *остаточными* ребрами, допускают положительный поток.

Заметим, что остаточное ребро (u, v) не обязано быть ребром сети G. Такое ребро из u в v появляется, когда $f(u, v) < 0$, то есть когда имеется поток вещества в обратном направлении.

Если ребро (u, v) принадлежит остаточной сети, то хотя бы одно из ребер (u, v) или (v, u) было в исходной сети. Поэтому $|E_f| \le 2 \cdot |E|$.

Пример остаточной сети

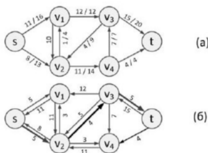


Рис. 7.5. а) сеть с заданным потоком, б) остаточная сеть

На рис. 7.5 а изображен поток f в сети G, на рис. 7.5 б – остаточная сеть G_f и выделен путь p, по которому можно еще пропустить 4 единицы. Заметим, что ребер (v_1, s) и (v_2, v_3) не было в исходной сети.

Пропускная способность ребра (v_1, v_2) равна 10, а пропускная способность ребра (v_2, v_1) равна 4, и $f(v_2, v_1) = 1$. Тогда по определению $c_f(v_2, v_1) = 4 - 1 = 3$. И, следовательно, $c_f(v_1, v_2) = 10 - (-1) = 11$, что и отображено на рис. 7.5 б. Пропускная способность ребра (v_3, v_2) равна 9, а ребра (v_2, v_3) равна 0, $f(v_3, v_2) = 4$. Поэтому $c_f(v_3, v_2) = 9 - 4 = 5$, $c_f(v_2, v_3) = 0 - (-4) = 4$.

Остаточная сеть G_f является сетью с пропускными способностями C_f.

Пусть имеется сеть $G = (V, E)$ и два потока f_1 и f_2 на ней.

Суммой потоков f_1 и f_2 называется функция, отображающая $V \times V$ в R:

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad (1)$$

Лемма 7.2

Пусть $G = (V, E)$ – сеть с истоком s и стоком t, а f – поток в ней.

Пусть G_f – остаточная сеть сети G, порожденная потоком f, а f' – поток в G_f. Тогда сумма потоков $f + f'$, определяемая уравнением (1), является потоком в сети G, и величина этого потока равна $|f + f'| = |f| + |f'|$.

Доказательство

Сначала докажем, что $f + f'$ будет потоком.

1) Проверим условие, связанное с *ограниченной пропускной способностью*. Заметим, что $f'(u,v) \le c_f(u,v)$ для всех $u, v \in V$. Поэтому
$$(f + f')(u, v) = f(u, v) + f'(u, v) \le f(u, v) + (c(u, v) - f(u, v)) = c(u, v).$$
2) *Антисимметричность*. Для всех $u, v \in V$ выполнено $(f + f')(u, v) = f(u,v) + f'(u,v) = -f(v,u) - f'(v,u) = -(f(v,u) + f'(v,u)) = -(f + f')(u, v)$.
3) *Закон сохранения потока*. Для всех $u \in V \setminus \{s, t\}$ выполнено:
$(f + f')(u,V) = f(u,V) + f'(u,V) = f(u,V) + f'(u, V) = 0 + 0 = 0.$

Наконец, найдем величину суммарного потока:

$$|f+f'| = (f+f')(s, V) = f(s,V) + f'(s,V) = |f| + |f'|.$$

7.4 Дополняющие пути

Пусть f – поток в сети $G = (V, E)$. *Дополняющим путем* называется простой путь из истока s в сток t в остаточной сети G_f.

Из определения остаточной сети вытекает, что по всем ребрам дополняющего пути можно переслать еще сколько-то вещества, не превысив пропускную способность ребра.

Максимальная величина, на которую можно увеличить поток вдоль каждого ребра дополняющего пути p, называется *остаточной пропускной способностью* пути p: $c_f = \min \{ c_f(u, v): (u, v) \in p \}$.

Лемма 7.3

Пусть f – поток в сети $G = (V, E)$ и p – дополняющий путь в G_f.

Определим функцию $f_p : V \times V \to R$:

$$f_p(u,v) = \begin{cases} c_f(p), & \text{если } (u,v) \in p, \\ -c_f(p), & \text{если } (v,u) \in p, \\ 0, & \text{в остальных случаях.} \end{cases} \qquad (2)$$

Тогда f_p является потоком в сети G_f и его величина составляет

$$|f_p| = c_f(p) > 0.$$

Без доказательства

Следствие 7.3.1

Пусть f – поток в сети $G = (V, E)$, а p – дополняющий путь в сети G_f, заданный равенством (2). Тогда функция $f' = f + f_p$ является потоком в сети G величины $|f'| = |f| + |f_p| > |f|$.

Доказательство вытекает из лемм 16.2 и 16.3.

Следовательно, если к потоку f добавить поток f_p, то в сети G получится поток с большим значением.

Пример

Рассмотрим остаточную сеть на 7.5 б и дополняющий путь, выделенный на этом рисунке жирными стрелками: (s, v_2), (v_2, v_3), (v_3, t). По нему можно пропустить поток величиной в 4 единицы. Тогда в результате получим сеть, показанную на рис. 7.6 в, и остаточную сеть, порожденную этим потоком (рис. 7.6 г).

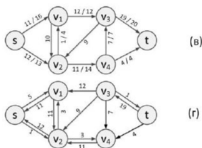


Рис. 7.6. в) сеть с заданным потоком, г) остаточная сеть

7.5 Разрезы в сетях

Разрезом сети $G = (V, E)$ называется разбиение множества V на две части S и $T = V \setminus S$, для которых $s \in S$ и $t \in T$.

Пропускной способностью разреза (S, T) является $c(S, T)$ – сумма пропускных способностей пересекающих разрез ребер.

Для заданного потока f величина чистого потока через разрез (S, T) определяется как $f(S, T)$ по пересекающим разрез ребрам.

Минимальным разрезом является разрез, пропускная способность которого среди всех разрезов данной сети минимальна.

На рис. 7.7 изображен разрез сети, где $S = \{s, v_1, v_2\}$, $T = \{v_3, v_4, t\}$.

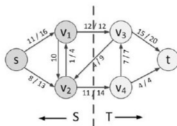


Рис. 7.7. Разрез сети через ребра (v_1, v_3) и (v_2, v_4)

Пропускная способность этого разреза равна

$$c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$$

Чистый поток через разрез равен

$$f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) = 12 + (-4) + 11 = 19.$$

Следует обратить внимание, что чистый поток через разрез, в отличие от пропускной способности разреза, может содержать отрицательные потоки между вершинами. Чистый поток через разрез (S, T) составляется из

положительных потоков в обоих направлениях, при этом положительный поток из S в T прибавляется, а положительный поток из T в S вычитается.

Рассмотрим другой разрез этой же сети (рис. 7.8), в котором

$$S = \{s, v_1, v_2, v_4\},\ T = \{v_3, t\}.$$

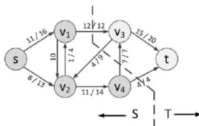


Рис. 7.8. Разрез сети через ребра (v_1, v_3) и (v_4, t)

Чистый поток через него равен

$$f(v_1, v_3) + f(v_2, v_3) + f(v_4, v_3) + f(v_4, t) = 12 + (-4) + 7 + 4 = 19,$$

пропускная способность разреза равна

$$c(v_1, v_3) + c(v_4, v_3) + c(v_4, t) = 12 + 7+ 4 = 23.$$

Лемма 17.4

Пусть f – поток в сети G с истоком s и стоком t, а (S, T) разрез сети G.

Тогда чистый поток через разрез (S, T) равен $f(S, T) = |f|$.

Без доказательства

Следствие 7.4.1

Величина любого потока f в сети G не превосходит пропускной способности любого разреза сети G.

Доказательство

$$|f| = f(S, T) = \sum_{u\in S}\sum_{y\in T} f(u, v) \le \sum_{u\in S}\sum_{y\in T} c(u, v) = c(S, T)$$

Теорема 7.1 о максимальном потоке и минимальном разрезе

Пусть f – поток в сети $G = (V, E)$ с истоком s и стоком t.

Тогда следующие утверждения эквивалентны:

1) Поток f является потоком максимальной величины в сети G.
2) Остаточная сеть G_f не содержит дополняющих путей.
3) Для некоторого разреза (S, T) сети G выполнено равенство $|f| = c(S, T)$. В этом случае разрез является минимальным, то есть имеет минимально возможную пропускную способность.

Доказательство

Методом от противного покажем, что из утверждения 1 следует утверждение 2. Пусть f является потоком максимальной величины, но остаточная сеть G_f содержит дополняющий путь p. Рассмотрим сумму потоков $f + f_p$,

где f_p задается равенством (2). По следствию 7.3.1 эта сумма потоков является потоком в сети G, величина которого больше $| f |$, что противоречит утверждению 1.

Покажем, что из утверждения 2 следует утверждение 3. Пусть в сети G_f нет пути из истока s в сток t. Рассмотрим множество $S = \{v \in V$: в G_f существует путь из s в $v\}$. Положим $T = V \setminus S$. Очевидно, что $s \in S$, а $t \in T$, так как в G_f нет пути из s в t. Поэтому пара (S, T) – разрез. Ни для каких $u \in S$ и $v \in T$ ребро (u, v) не принадлежит E_f , так как в противном случае вершина v попала бы в S. Поэтому $f(u, v) = c(u, v)$. Из леммы 7.4 следует, что $| f | = f(S, T) = c(S, T)$.

Для доказательства того, что из утверждения 3 следует утверждение 1, достаточно применить следствие 7.4.1: для любого разреза (S, T) величина любого потока $| f | \le c(S, T)$. Поэтому из равенства $| f | = c(S, T)$ следует, что поток f максимален.

7.6 Метод Форда-Фалкерсона

Для решения задачи о максимальном потоке рассмотрим метод Форда-Фалкерсона. Этот метод допускает несколько реализаций с различным временем выполнения. Он основывается на трех важных понятиях: остаточные сети, дополняющие пути и разрезы.

Метод Форда-Фалкерсона является итеративным. Вначале величине потока присваивается значение 0. На каждой итерации величина потока увеличивается. Для этого определяется дополняющий путь, и он используется для увеличения потока. Этот процесс продолжается до тех пор, пока невозможно отыскать дополняющий путь.

Алгоритм 7.1. Алгоритм Форда-Фалкерсона (общий случай)

Вход: Сеть $(G= (V, E), s, t)$, где s, t – исток и сток, соответственно.
Выход: Величина максимального потока.
Метод:

```
начало
    |f| ← 0
    пока (существует дополняющий путь p) выполнить
        дополнить f вдоль p
    конец цикла
    выдать |f|
конец
```

Конец алгоритма 7.1

Для детализации алгоритма 7.1 и его реализации будем хранить текущие значения потока между вершинами u и v в массиве $f [u, v]$. Тогда алгоритм будем выглядеть так, как показано ниже.

Алгоритм 7.1. Алгоритм Форда-Фалкерсона (уточнение)

Вход: Сеть ($G = (V, E)$, s, t), где s, t – исток и сток, соответственно.

Выход: Величина максимального потока.

Метод:

начало

  для всех $(u, v) \in E$ **выполнить**

    $f[u, v] \leftarrow 0$

    $f[v, u] \leftarrow 0$

  конец цикла

  пока (в G_f существует путь p из s в t) **выполнить**

    $c_f(p) \leftarrow \min\{ c_f(u, v) : (u, v)$ входит в p }

    для всех $(u, v) \in p$ **выполнить**

      $f[u, v] \leftarrow f[u, v] + c_f(p)$

      $f[v, u] \leftarrow -f[u, v]$

    конец цикла

  конец цикла

конец

Конец алгоритма 7.1

Время работы метода Форда-Фалкерсона зависит от того, как ищется дополняющий путь p. Если находить его при помощи поиска в ширину, то алгоритм работает за полиномиальное время.

Рассмотрим случай, когда пропускные способности ребер – целые числа. На инициализацию затрачивается время $O(|E|)$. Цикл **пока** выполняется не более $|f^*|$ раз, где f^* – максимальный поток. Поиск дополняющего пути в остаточной сети займет время $O(|E|)$. Поэтому время работы данного алгоритма будет $O(|E|\cdot|f^*|)$.

Если использовать поиск в ширину, то путь p будет кратчайшим из дополняющих путей, длину каждого ребра считаем равной 1. Такая реализация метода Форда-Фалкерсона называется алгоритмом Эдмондса-Карпа. Время работы алгоритма Эдмондса-Карпа равно $O(|V|\cdot|E|^2)$.

7.6.1 Пример

Рассмотрим на примере из [5] работу алгоритма 7.1. На рис. 7.9 (а) показана остаточная сеть, в которой жирными стрелками выделен дополняющий путь p_1. Минимальная пропускная способность в этом пути у ребра (v_3, t). По этому пути можно пропустить поток $f_{p1} = 4$. На рис. 7.9 (б) изображена сеть с потоком f_{p1}.

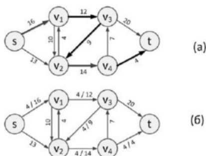


Рис. 7.9. Сеть: а) выделен дополняющий путь, б) указан поток $f_{p1} = 4$

На рис. 7.10 показана остаточная сеть, порожденная по потоку f_{p1}. В ней выделен дополняющий путь p_2, по которому можно пропустить поток величины 7. Заметим, что из вершины v_3 уже нет ребра в вершину t. На второй сети этого рисунка показан поток $f_p = f_{p1} + f_{p2} = 4 + 7 = 11$.

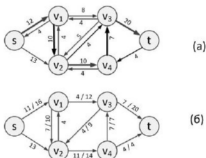


Рис. 7.10. Остаточная сеть: а) дополняющий путь, б) поток $f_p = 11$

На рис. 7.11 показана остаточная сеть, порожденная по потоку $f_p = 11$. В ней выделен дополняющий путь p_3, по которому можно пропустить поток величины 8. На второй сети этого рисунка показан поток

$f_p = f_{p1} + f_{p2} + f_{p3} = 4 + 7 + 8 = 19$.

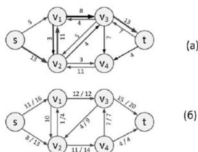


Рис. 7.11. Остаточная сеть: а) дополняющий путь, б) поток $f_p = 19$

На рис. 7.12 показана остаточная сеть, порожденная по потоку $f_p = 19$. В ней выделен дополняющий путь p_4, по которому можно пропустить поток величины 4. На второй сети этого рисунка показан поток

$$f_p = f_{p1} + f_{p2} + f_{p3} + f_{p4} = 4 + 7 + 8 + 4 = 23.$$

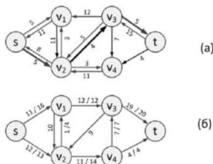


Рис. 7.12. Остаточная сеть: а) дополняющий путь, б) поток $f_p = 23$

Остаточная сеть, порожденная по потоку $f_p = 23$, показана на рис. 7.13.

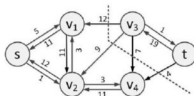


Рис. 7.13. Остаточная сеть, порожденная потоком $f_p = 23$

Отсечем в этой сети часть вершин, до которых имеется путь из источника. Это будут вершины: v_1, v_3, v_4. Таким образом, мы получили минимальный разрез, при этом множество $S = \{s,\ v_1,\ v_3,\ v_4\}$, множество $T = \{v_2,\ t\}$.

Пропускная способность этого разреза определяется ребрами сети с рис. 7.2: $c(v_1, v_3) + c(v_4, v_3) + c(v_4, t) = 23$.

7.7 Задача о максимальном паросочетании в двудольном графе

Пусть $G = (V, E)$ – неориентированный граф.

Паросочетанием назовем подмножество ребер $M \subseteq E$, такое, что для всех вершин $v \in V$ в M содержится не более одного ребра, инцидентного v. Вершина v является *связанной* паросочетанием M, если в M имеется ребро, инцидентное v, иначе вершина v называется *открытой*.

Максимальное паросочетание – это паросочетание M, содержащее максимально возможное число ребер: $|M| \geq |M'|$ для любого паросочетания M'.

Рассмотрим паросочетания в *двудольных* графах. В них множество V можно разбить на два непересекающихся подмножества L и R, и любое ребро из E соединяет некоторую вершину из L с некоторой вершиной из R. Например, L – женихи, R – невесты, наличие ребра (u, v) означает, что u и v согласны стать супругами. Максимальное паросочетание будет доставлять ЗАГСу больше всего работы.

Используя метод Форда-Фалкерсона, можно определить максимальное паросочетание в двудольном графе $G = (V, E)$ за время, полиномиально зависящее от $|V|$ и $|E|$.

Рассмотрим сеть $G' = (V', E')$, построенную из графа G следующим образом:

– в граф добавлены две новые вершины: исток (s) и сток (t):

$$V' = V \cup \{s, t\};$$

– построено множество ребер

$$E' = \{(s,u)\colon u \in L\} \cup \{(u, v)\colon u \in L, v \in R, (u, v) \in E\} \cup \{(v, t)\colon v \in R\};$$

– пропускная способность каждого ребра задана равной единице.

Поток f в сети $G = (V, E)$ называется *целочисленным*, если все значения $f(u, v)$ – целые.

Лемма 7.5

Пусть $G = (V, E)$ – двудольный граф с долями L и R, и $G' = (V, E')$ – соответствующая сеть. Пусть M – паросочетание в G.

Тогда существует целочисленный поток в G', величина которого $|f| = |M|$. Справедливо и обратное утверждение: если f – целочисленный поток в G', то в G' найдется паросочетание M с мощностью $|M| = |f|$ элементов.

Без доказательства

Теорема 7.2 о целочисленном потоке

Если пропускные способности всех ребер – целые числа, то максимальный поток, найденный алгоритмом Форда - Фалкерсона, будет целочисленным.

Без доказательства

Следствие 7.2.1

Число ребер в максимальном паросочетания M в двудольном графе G равно значению максимального потока в сети G'.

Пример

Рассмотрим двудольный граф и соответствующую сеть G' на рис. 7.14. Пропускная способность любого ребра этой сети равна единице. Установим поток по выделенным ребрам равным единице, по остальным – нулю. Выделенные ребра, соединяющие вершины из L с вершинами из R, соответствуют максимальному паросочетанию в двудольном графе.

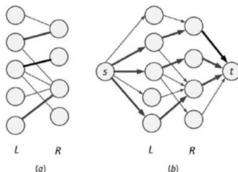


Рис. 7.14. Двудольный граф и соответствующая сеть G'

Таким образом, чтобы найти максимальное паросочетание в двудольном графе G, нам достаточно применить метод Форда - Фалкерсона и найти максимальный поток в соответствующей сети G'. Никакое паросочетание в двудольном графе не может содержать более min $(|L|, |R|) = O(|V|)$ ребер. Поэтому значение максимального потока в G' равно $O(|V|)$. Следовательно, время работы алгоритма Форда - Фалкерсона равно $O(|V|\cdot|E|)$.

7.8 Алгоритм проталкивания предпотока

Рассмотрим подход к вычислению максимального потока, основанный на «проталкивании предпотока». На этом методе основаны реальные реализации алгоритмов поиска максимального потока. Алгоритмы проталкивания предпотока исследуют сеть локально, обрабатывают вершины по одной, рассматривая только соседей данной вершины в остаточной сети. Кроме того, в ходе их исполнения не обеспечивается свойство сохранения потока.

Определим *предпоток* как функцию $f : V \times V \to R$, которая антисимметрична, удовлетворяет ограничениям, связанным с пропускными способностями, а также ослабленному закону сохранения потока: $f(V, u) \geq 0$ для всех вершин $u \in V \setminus \{s\}$. Таким образом, в каждой вершине u (кроме истока) есть некоторый неотрицательный *избыток* $e(u) = f(V, u)$.

Вершина, отличная от s, с положительным избытком называется *переполненной.*

Важную роль играет целочисленный параметр – *высота* вершины. Будем мысленно представлять, что в процессе работы алгоритма вершина может подниматься вверх. Высота вершины определяет, куда можно направить избыток жидкости. Высота истока всегда равна $|V|$, а стока – нулю. Все остальные вершины изначально находятся на высоте 0, и в процессе работы алгоритма поднимаются.

Для начала из истока вниз отправляется столько жидкости, сколько позволяют пропускные способности выходящих из истока «труб». Это количество равно пропускной способности разреза $(s, V \setminus s)$. Избыток жидкости, возникающий в соседних с истоком вершинах, затем будет направлен дальше.

Рассматривая какую-либо вершину u в ходе работы алгоритма, можно обнаружить, что в ней имеется избыток жидкости, но все трубы, по которым еще можно отправить жидкость из u в *ненасыщенные* трубы ведут в вершины той же или большей высоты. В этом случае возможно выполнение операции, называемой *подъемом вершины u.*

В результате операции подъема вершина u оказывается расположенной на единицу выше самого низкого из тех ее соседей, в которых ведет ненасыщенная труба. Таким образом, вершина u поднимается ровно на такую высоту, чтобы появилась ненасыщенная труба, ведущая вниз.

В итоге в сток приходит максимально возможное количество жидкости. При этом предпоток может еще не быть потоком. Продолжая подъем вершин, которые могут стать выше истока, мы постепенно отправим избыток обратно в исток, что означает сокращение потока жидкости от истока. Тем самым предпоток становится потоком, который окажется максимальным.

7.8.1 Высотная функция

В алгоритме проталкивания предпотока выполняются две основные операции: проталкивание предпотока и подъем вершины. Их применение зависит от высот вершин.

Пусть $G = (V, E)$ – сеть с истоком s и стоком t, а f – предпоток в G. Функция $h : V \rightarrow N$ называется *высотной функцией* для предпотока f, если для любого ребра $(u, v) \in E_f$ выполнено:

$h(s) = |V|$, $h(t) = 0$, $h(u) \leq h(v) + 1$.

Лемма 7.6

Пусть f – предпоток в сети $G = (V, E)$, h – высотная функция. Тогда если для вершин $u, v \in V$, выполнено неравенство $h(u) > h(v) + 1$, то остаточная сеть не содержит ребра (u, v).

Без доказательства

Таким образом, по круто идущим вниз трубам идет максимально возможный поток.

7.8.2 Операция проталкивания предпотока

Операция проталкивания предпотока, которую реализует процедура *Push_flow* (u, v), применима, если:

1) вершина u переполнена: $e(u) > 0$;
2) ребро (u, v) не насыщено: $c_f(u, v) > 0$;
3) $h(u) = h(v) + 1$.

Предполагается, что остаточные пропускные способности при заданных f и c можно вычислить за фиксированное время.

Условие $h(u) = h(v) + 1$ гарантирует, что мы направляем дополнительный поток лишь по ребрам, идущим вниз с единичной разницей высот.

В описании процедуры используется временная переменная $d_f(u, v)$, в которой хранится количество потока, который можно протолкнуть из вершины u в вершину v.

```
процедура Push_flow (u, v)
    если e(u) > 0 и c_f (u, v) > 0 и h(u) = h(v) + 1 то
        начало
            d_f (u,v) ← min(e(u), c_f (u,v)
            f[u,v] ← f[u,v] + d_f (u,v)
            f[v, u] ← – f[u, v]
            e[u] ← e[u] – d_f (u,v)
            e[v] ← e[v] + d_f (u,v)
        конец
конец процедуры
```

Проталкивание называется *насыщающим*, если в результате ребро (u,v) становится *насыщенным*, то есть если $c_f\,(u,v)$ обращается в ноль. В противном случае проталкивание считают *ненасыщающим*.

Насыщенное ребро исчезает из остаточной сети.

Итак, если у вершины имеется сосед, который на единицу ниже ее, то можно выполнить операцию проталкивания, но нельзя выполнить подъем. Если все соседи не ниже рассматриваемой вершины, то проталкивание выполнить нельзя, а подъем – можно, после чего возможно применение операции проталкивания.

7.8.3 Операция поднятия вершины

Операция поднятия вершины, которую реализует процедура *Lift* (u), применима, если:

1) вершина u переполнена: $e(u) > 0$;
2) для любого ребра $(u,v)\in E_f$ выполнено: $h(u) \le h(v)$.

Операция *Lift* (u) поднимает переполненную вершину u на максимальную высоту, которая допустима по определению высотной функции: а именно, высота вершины превосходит высоту соседа в остаточной сети не более чем на 1.

процедура *Lift* (u)
  если $e(u) > 0$ **и** для любого ребра $(u, v)\in E_f$ выполнено: $h(u) \le h(v)$ **то**
    $h[u] \leftarrow 1 + \min\{\, h[v]\colon (u, v)\in E_f \,\}$
конец процедуры

Заметим, если вершина u переполнена, то в E_f найдется, по крайней мере, одно ребро, выходящее из u. Это следует из того, что если $f[V, u] = e[u] > 0$, то существует, по крайней мере, одна такая вершина v, для которой $f(v, u) > 0$.

Отсюда, $c_f\,(u, v) = c(u, v) - f\,[u, v] = c(u, v) + f\,[v, u] > 0$. Это значит, что $(u, v) \in E_f$.

7.8.4 Описание алгоритма

В универсальном алгоритме проталкивания предпотока задается начальный поток, устанавливается начальный избыток в каждой вершине, определяются начальные значения высотной функции.

Далее определяется предпоток по формуле:

$$f(u,v) = \begin{cases} c(u,v), \text{если } u = s, \\ -c(v,u), \text{если } v = s, \\ 0, \text{в остальных случаях.} \end{cases}$$

Затем следует ряд операций проталкивания или подъема, выполняемых без определенного порядка.

Алгоритм 7.2. Проталкивание предпотока

Вход: сеть (G, s, t), где s, t – исток и сток, соответственно.
Выход: величина максимального потока.
Метод:

начало
  для всех $(u \in V)$ **выполнить**
    $h[u] \leftarrow 0$
    $e[u] \leftarrow 0$
  конец цикла
  для всех $(u, v) \in E$ **выполнить**
    $f[u, v] \leftarrow 0$ | *По всем ребрам поток равен* 0
    $f[v, u] \leftarrow 0$
    $h[s] \leftarrow |V|$
  конец цикла
  для всех u: $(s, u) \in E$ **выполнить**
    $f[s, u] \leftarrow c(s, u)$ | *Поток по ребру, выходящему из истока s,*
                  | *равен пропускной способности этого ребра*
    $f[u, s] \leftarrow -c(s, u)$
    $e[u] \leftarrow c(s, u)$ | *В смежной с истоком вершине – избыток*
  конец цикла
  пока существует применимая операция проталкивания или подъема
  выполнить
    выбрать одну из операций *Push_flow* (u, v) **или** *Lift* (u)
    и применить ее
  конец цикла
конец

Конец алгоритма 7.2

Функция h, заданная в алгоритме 7.2, является высотной функцией, так как все ребра (u, v), для которых $h[u] > h[v] + 1$, выходят только из истока, но они насыщены и их нет в остаточной сети.

Лемма 7.7

Пусть f – предпоток в сети $G = (V, E)$. Пусть h – высотная функция для f и вершина u переполнена. Тогда в u возможно либо проталкивание, либо подъем.

Доказательство

Поскольку h – высотная функция, то $h(u) \le h(y) +1$ для любого остаточного ребра (u, v). Если в u невозможно проталкивание, то для всех остаточных ребер (u, v) выполнено неравенство $h(u) < h(v) + 1$, из чего следует, что $h(u) \le h(v)$, и в вершине u возможен подъем.

7.8.5 Корректность метода

Если алгоритм остановится, то предпоток f в этот момент будет максимальным потоком. Алгоритм действительно остановится.

Лемма 7.8

При исполнении алгоритма 7.2 высота $h[u]$ любой вершины $u \in V$ может только возрастать.
Без доказательства

Лемма 7.9

Во время выполнения алгоритма 7.2 функция h остается высотной функцией.
Без доказательства

Лемма 7.10

Пусть $G = (V, E)$ – сеть с истоком s и стоком t. Пусть f – предпоток в G, а h – высотная функция для f. Тогда в остаточной сети G_f не существует пути из истока в сток.
Без доказательства

Теорема 7.3

Если алгоритм 7.2, примененный к сети $G = (V, E)$ с истоком s и стоком t останавливается, то получающийся предпоток f, будет максимальным потоком для G.

Пояснение. В момент остановки переполненных вершин в сети нет. Значит, в этот момент предпоток является потоком. Функция h будет высотной функцией, и потому в остаточной сети G_f нет пути из s в t. По теореме о максимальном потоке и минимальном разрезе поток f максимален.
Без доказательства

Лемма 7.11

Пусть $G = (V, E)$ – сеть с истоком s и стоком t, а f – предпоток в G. Тогда для любой переполненной вершины u найдется простой путь из u в s в остаточной сети G_f.
Без доказательства

Лемма 7.12

При исполнении алгоритма 7.2 высота любой вершины $v \in V$ не превосходит $(2 \cdot |V| - 1)$.
Без доказательства

Следствие 7.12.1

При исполнении алгоритма 7.2 общее число операций подъема не превосходит $2 \cdot |V|^2$.

Лемма 7.13

При исполнении алгоритма 7.2 количество насыщающих проталкиваний не превосходит $2 \cdot |V| \cdot |E|$.
Без доказательства

Теорема 7.4

Общее число операций подъема и проталкивания при исполнении алгоритма 7.2 на сети $G = (V, E)$ равно $O(|V|^2 \cdot |E|)$.
Без доказательства

7.8.6 Пример

На рис. 7.15 представлена сеть с рис. 7.2, в которой все вершины, кроме стока расположены на высоте 0, сток – на высоте 6. Более точно, на этом рисунке отображен момент исполнения действий алгоритма 7.2 после задания начальных значений и после того, как в вершины v_1 и v_2 поступило 16 и 13 единиц жидкости, соответственно. Избыток жидкости в этих вершинах составил e_1= 16 и e_2=13 единиц.

Ребра, по которым пропущен поток, в нашем случае, это (s, v_1) и (s, v_2), будем изображать на рисунках пунктирными линиями.

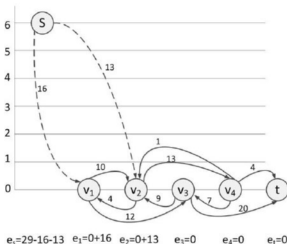


Рис. 7.15. Сеть, соответствующая сети с рис. 7.2 с указанием уровней вершин

Продолжая действия алгоритма 7.2, можно добиться того, что в стоке окажется максимально возможное количество жидкости. При этом некоторые вершины могут оставаться переполненными. При дальнейшем подъеме, они окажутся выше истока. Постепенно проталкивая из них избыток обратно в исток, добьемся, что предпоток станет потоком. Продемонстрируем дальнейшие шаги алгоритма на примере сети с рис. 7.15.

Так как проталкивание оказалось насыщающим, то ребра (s, v_1) и (s, v_2) в остаточной сети на рис. 7.16 отсутствуют, но появляются ребра (v_1, s) и (v_2, s) с пропускными способностями 16 и 13, соответственно. Все вершины остаточной сети, кроме источника, находятся на одном уровне. Следовательно, несмотря на то что имеются переполненные вершины, выполнить операцию проталкивания нельзя.

Поднимем, например, вершину v_2 и протолкнем из нее 13 единиц жидкости в вершину v_4.

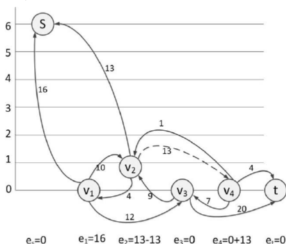


Рис. 7.16. Поднятие вершины v_2

В результате в остаточной сети ребро (v_2, v_4) исчезает, пропускная способность ребра (v_4, v_2) становится равной 14, что отображено в остаточной сети на рис. 7.17. Избыток жидкости в вершине v_4: $e(v_4) = 13$.

Таким образом, избыток жидкости имеется в вершинах v_1 и v_4. Поднимем вершину v_4 на одну позицию. Из нее можно протолкнуть 7 единиц жидкости в вершину v_3 и 4 единицы в вершину t. Затем поднимем вершину v_1 и протолкнем из нее 12 единиц в вершину v_3. Эти действия алгоритма отображены на рис. 7.17. Остаточная сеть показана на рис. 7.18.

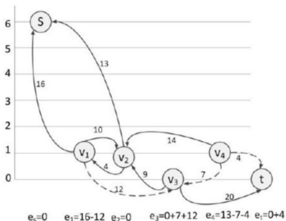


Рис. 7.17 Поднятие вершин v_1 и v_4 и проталкивание предпотока

Избыток жидкости в вершине v_3 равен 19. Поднимем вершину v_3 на одну позицию и протолкнем 19 единиц жидкости в вершину в вершину t (рис. 7.18).

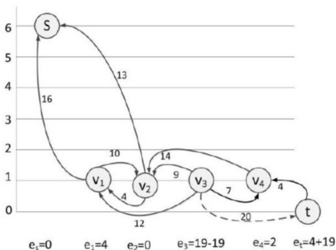


Рис. 7.18. Поднятие вершины v_3 и проталкивание предпотока

Остаточная сеть после этого проталкивания показана на рис. 7.19.

На этом шаге алгоритма $e(v_1) = 4$, $e(v_4) = 2$. Поднимем по очереди вершины v_1 и v_4. В вершину v_2 можно протолкнуть из вершины v_4 2 единицы жидкости, а из вершины v_1 – 4 единицы (рис. 7.19).

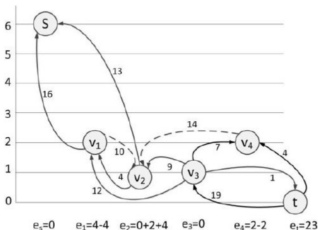


Рис. 7.19. Поднятие вершин v_1 и v_4 и проталкивание предпотока

В результате этих проталкиваний строится остаточная сеть, которая показана на рис. 7.20.

Итак, $e(v_2)$=6. Поднимем вершину v_2 на высоту 3 и отправим в v_1 6 единиц (рис. 7.20). Остаточная сеть показана на рис. 7.21.

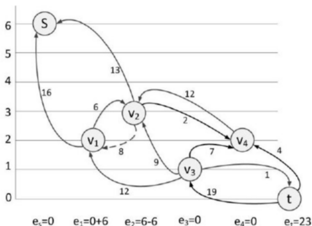


Рис. 7.20. Поднятие вершины v_2 и проталкивание предпотока

Избыток жидкости в вершине v_1 составляет 6 единиц. В остаточной сети имеется дуга (v_1, v_2). Поднимем вершину v_1 на высоту 4 и перельем 6 единиц жидкости в вершину v_2 (рис. 7.21).

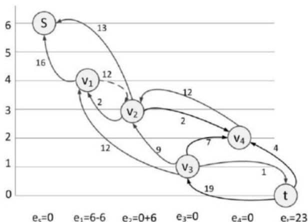


Рис. 7.21. Поднятие вершины v_1 и проталкивание предпотока

На этом шаге алгоритма избыток жидкости, равный 6 единицам, имеется в вершине v_2. По дуге (v_2, v_4) протолкнем 2 единицы (рис. 7.22). После этого из вершины v_2 невозможно протолкнуть остаток жидкости ни в какие соседние с ней вершины.

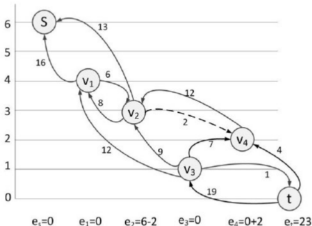


Рис. 7.22. Проталкивание предпотока в вершину v_4

Поднимем вершину v_4 на высоту 4 и протолкнем 2 единицы жидкости в вершину в вершину v_2 (рис. 7.23). В результате этого проталкивания построена остаточная сеть на рис. 7.24.

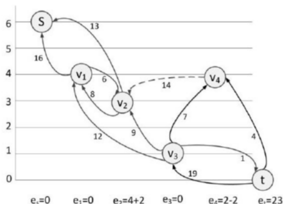


Рис. 7.23. Поднятие вершины v_4 и проталкивание предпотока

Избыток жидкости в вершине v_2 равен 6. В остаточной сети из нее имеются дуги в вершины v_1 и v_4, которые находятся на высоте 4. Поднимем вершину v_2 на высоту 5 и протолкнем 6 единиц жидкости в вершину v_1 (рис. 7.24).

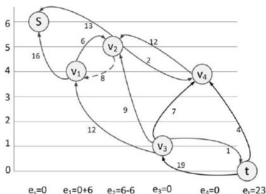


Рис. 7.24. Поднятие вершины v_2 и проталкивание предпотока

Теперь избыток жидкости равный 6 единицам имеется в вершине v_1. В остаточной сети из вершины v_1 имеются дуги в вершины s и v_2, которые находятся на высоте 6 и 5, соответственно. Поднимем вершину v_1 на высоту

6 и протолкнем 6 единиц жидкости в вершину v_2 (рис. 7.25). Остаточная сеть в результате этого проталкивания показана на рис. 7.26.

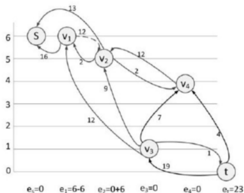


Рис. 7.25. Поднятие вершины v_1 и проталкивание предпотока

В результате последнего проталкивания в вершине v_2 имеется избыток, равный 6 единицам. Поскольку вершина v_2 находится на высоте 5, то из нее можно протолкнуть 2 единицы в вершину v_4. Это действие отображено на рис. 7.26, как и ранее, пунктирной дугой. Остаточная сеть, построенная после этого проталкивания, показана на рис. 7.27.

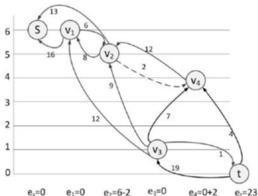


Рис. 7.26. Проталкивание предпотока из вершины v_2 в вершину v_4

На этом шаге избыток жидкости имеется в вершинах v_2 и v_4, 4 и 2 единицы, соответственно. В остаточной сети на рис. 7.27 показано, что из вершины v_4 имеется выходящая дуга только в вершину v_2, а из v_2 – в вершины s и v_1. Но проталкивание невозможно, так как v_2 и v_4 находятся ниже своих

соседей. Поднимем вершину v_4 на высоту 6 и протолкнем 2 единицы в вершину v_2 (рис. 7.27). Остаточная сеть показана на рис. 7.28.

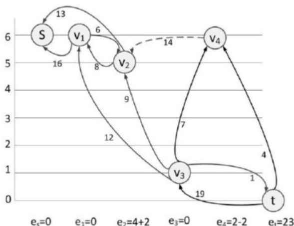


Рис. 7.27. Поднятие вершины v_4 и проталкивание предпотока

И, наконец, из вершины v_2 протолкнем весь избыток в вершину s (рис. 7.28).

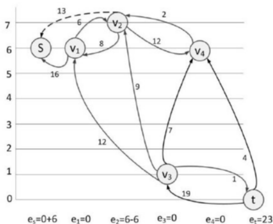


Рис. 7.28. Поднятие вершины v_2 и проталкивание предпотока

Таким образом, постепенно из вершин избыток жидкости отправлен обратно в исток, тем самым предпоток стал потоком, в стоке оказалось максимальное количество жидкости.

ЗАКЛЮЧЕНИЕ

В данном учебном пособии описаны динамические структуры данных, абстрактные типы и эффективные алгоритмы их обработки.

Структура данных – это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию. Ни одна структура не является универсальной и не может подходить для всех целей, поэтому важно знать преимущества и ограничения, присущие некоторым из них.

В 1968 году Кнут опубликовал первый из трех томов, объединенных названием «Искусство программирования» (*The Art of Computer Programming*). Этот том стал введением в современные компьютерные алгоритмы с акцентом на анализе времени их работы, а весь трехтомник до сих пор остается интереснейшим и ценным пособием по многим темам.

Книга А. Ахо, Д. Хопкрофта, Д. Ульмана «Структуры данных и алгоритмы» [4] является прекрасным справочным пособием по элементарным структурам данных и алгоритмам, которые являются фундаментом современной методологии разработки программ.

Фундаментальный труд известных авторов Т. Кормена, Ч. Лейзерсона, Р. Ривеста, К. Штайна «Алгоритмы: построение и анализ» [5] является настольной книгой для студентов, специализирующихся в области компьютерных наук.

Идея балансировки деревьев поиска принадлежит советским математикам Г. М. Адельсону-Вельскому и Е. М. Ландису, предложившим в 1962 г. класс сбалансированных деревьев поиска, получивших название АВЛ - деревьев и описанных в разделе 7.6. В этом учебном пособии не описаны красно-черные деревья, предложенные Байером под названием «симметричные бинарные B-деревья» и представляющие собой одну из множества «сбалансированных» схем деревьев поиска, и более просты в реализации по сравнению с АВЛ - деревьями.

Глубоко не рассматривались также жадные алгоритмы, которые используются в задачах оптимизации и хорошо подходят для довольно широкого класса задач. Однако несколько жадных алгоритмов представлены в этом пособии, это алгоритмы поиска минимальных остовных деревьев, алгоритм Дейкстры для определения кратчайших путей из одного источника. За рамками данного учебного пособия остались хеш-таблицы, представляющие собой эффективную структуру данных для реализации словарей, и построение хеш-функций. Алгоритмы хеширования прекрасно изложены в [5].

Несмотря на то, что вышеперечисленные вопросы не рассматриваются в данном пособии, авторы надеются, что приведенный материал будет полезен широкому кругу читателей, заинтересованных в решении задач, связанных с искусством программирования.

СПИСОК ЛИТЕРАТУРЫ

1. *Вирт Н.* Алгоритмы и структуры данных / Н. Вирт. М.: Мир, 1989. 360с.
2. *Ахо А.* Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ / А. Ахо, Дж. Ульман : пер. с англ. Москва : Мир, 1977. 612 с.
3. *Ахо А.* Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман : пер. с англ. Москва : Мир, 1979. 536 с.
4. *Ахо А.* Структуры данных и алгоритмы / Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман : пер. с англ. : Уч. пос. М.: Издательский дом "Вильямс", 2000. 384 с.
5. *Кормен Т.* Алгоритмы: Построение и анализ / Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. 2-е издание : пер. с англ. М: Издательский дом "Вильямс", 2005. 1296 с.
6. *Подбельский В. В.* Программирование на языке Си : учеб. пособие / В. В. Подбельский, С. С. Фомин. 2-е доп. изд. М. : Финансы и статистика, 2007. 600 с.
7. *Харари Ф.*Теория графов / Ф. Харари. Москва: Мир, 1973. 300 с.
8. *Хэзфилд Р.* Искусство программирования на С: Фундаментальные алгоритмы, структуры данных и примеры приложений (пер. с англ.) / Ричард Хэзфилд, Лоуренс Кирби и др. : пер. с англ. Киев : ДиаСофт, 2001. 736 с. (Энциклопедия программиста)
9. *Цикоза В. А.* Методы программирования: представление и кодирование информации. Часть 1 : учеб. пособие / В. А. Цикоза, Т. Г. Чурина. Новосибирск: НГУ, 2003. 48 с.
10. *Цикоза В. А.* Методы программирования: перестановки, поиск, сортировки. Часть 2 : учеб. пособие / В. А. Цикоза, Т. Г. Чурина. Новосибирск: НГУ, 2006. 59 с.
11. *Всероссийская олимпиада школьников* 2004 года, разбор задач [Электронный ресурс]. http://informatics.mccme.ru/moodle/file.php/4/razbor-russia-2004-1.pdf
12. *Декартово_дерево* [Электронный ресурс]. http://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево