

Граф, как и большинство других математических объектов, может быть представлен на компьютере (сохранен в его памяти). Существуют несколько способов его интерпретации, вот наиболее известные из них:

- матрица смежности;
- матрица инцидентности;
- список смежности;
- список ребер.

Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Причем размеры этих массивов, зависят от количества вершин и/или ребер в конкретном графе. Так размер матрицы смежности $n \times n$, где n – число вершин, а матрицы инцидентности $n \times m$, n – число вершин, m – число ребер в графе.

Базовая терминология

| Рус | En | Описание |
|---------------|-------------|---|
| Вершина | vortex | Элемент графа |
| Узел | node | Тоже, что и вершина |
| Ребро | edge | Связь двух соседних вершин |
| Дуга | arc | Тоже, но в орграфе |
| Связь | link | Элемент графа (ребро или дуга) |
| Смежность | adjacent | О двух вершинах между которыми есть связь |
| Инцидентность | incident on | О ребре по отношению к вершине |
| Степень | degree | Число ребер инцидентных вершине |

Матрица смежности

Матрица смежности графа — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Прежде чем отобразить граф через матрицу смежности, рассмотрим простой пример такой матрицы (рис. 12.2).

| | | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 |

Рисунок 12.2

Это двоичная квадратная матрица, т. к. число строк в ней равно числу столбцов, и любой из ее элементов имеет значение либо 1, либо 0. Первая строка и первый столбец (не входят в состав матрицы, а показаны здесь для легкости ее восприятия) содержат номера, на пересечении которых находится каждый из элементов, и они определяют индексное значение последних. Имея в наличии лишь матрицу такого типа, несложно построить соответствующий ей граф (рис. 12.3).

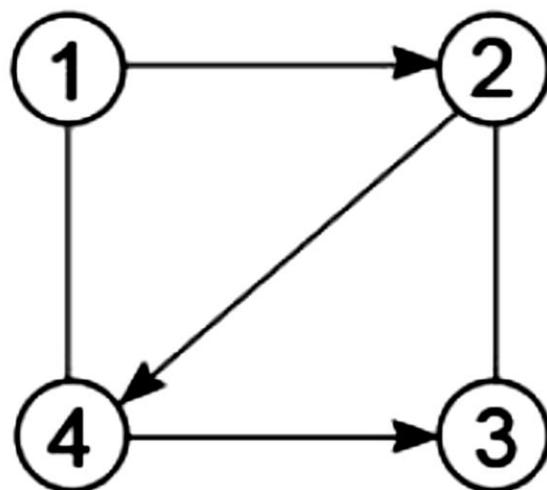


Рисунок 12.3

Таким образом, если из i в j существует ребро, то $A[i][j]:=1$, в противном случае $A[i][j]:=0$. Как видно, все элементы на главной диагонали равны нулю, это следствие отсутствия у графа петель.

В программе матрица смежности задается при помощи обычного двумерного массива, имеющего размерность $n \times n$, где n – число вершин графа. На языке C++, описать ее можно, например, так:

```
int graph[n][n] = {  
    {0, 1, 0, 1},  
    {0, 0, 1, 1},  
    {0, 1, 0, 0},  
    {1, 0, 1, 0}};
```

Матрица инцидентности

Две вершины u и v являются смежными, если в графе G существует ребро (u, v) в противном случае u и v независимы. Про ребро (u, v) также говорят, что оно инцидентно вершинам u и v .

Матрица инцидентности строиться похожему, но не по тому же принципу, что и матрица смежности. Так если последняя имеет размер $n \times n$, где n – число вершин, то матрица инцидентности – $n \times m$, здесь n – число вершин графа, m – число ребер. То есть теперь чтобы задать значение какой-либо ячейки, нужно сопоставить не вершину с вершиной, а вершину с ребром.

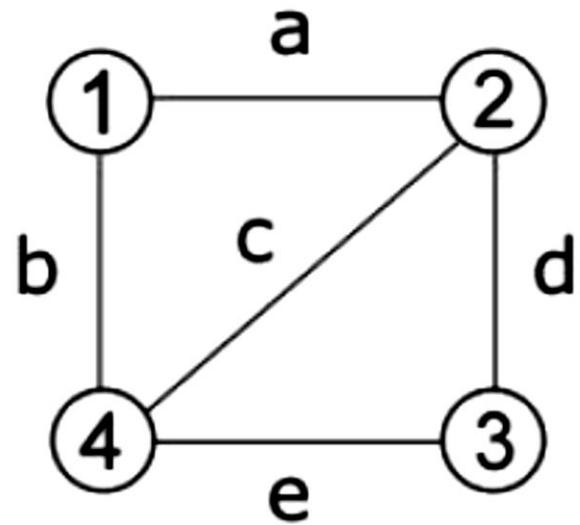
В каждой ячейки матрицы инцидентности *неориентированного графа* стоит 0 или 1, а в случае *ориентированного графа*, вносятся 1, 0 или -1. То же самое, но наиболее структурировано:

1. Неориентированный граф

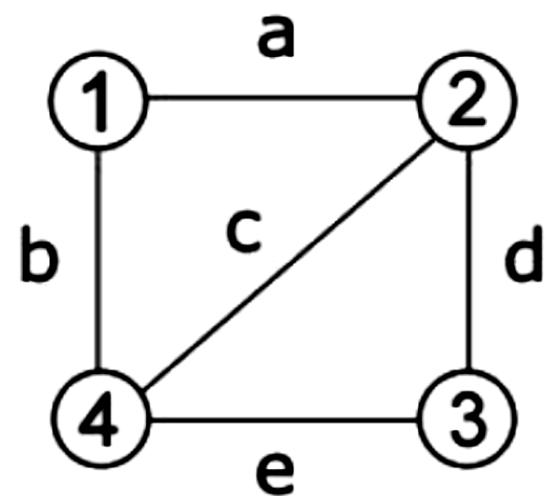
- a) 1 – вершина инцидентна ребру
- b) 0 – вершина не инцидентна ребру

2. Ориентированный граф

- a) 1 – вершина инцидентна ребру, и является его началом
- b) 0 – вершина не инцидентна ребру
- c) -1 – вершина инцидентна ребру, и является его концом

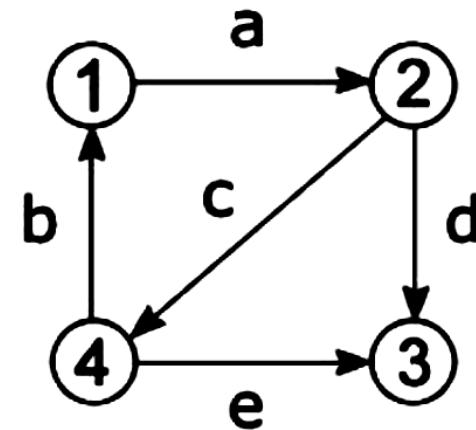


| | a | b | c | d | e |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 1 |



Ребра обозначены буквами от а до е, вершины – цифрами. Все ребра графа не направлены, поэтому матрица инцидентности заполнена положительными значениями.

| | a | b | c | d | e |
|---|----|----|----|----|----|
| 1 | 1 | -1 | 0 | 0 | 0 |
| 2 | -1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | -1 | -1 |
| 4 | 0 | 1 | -1 | 0 | 1 |



В программе матрица инцидентности задается, также как и матрица смежности, а именно при помощи двумерного массива. Его элементы могут быть инициализированы при объявлении, либо по мере выполнения программы.

Список смежности

По отношению к памяти списки смежности менее требовательны, чем матрицы смежности, для их хранения нужно $O(|V| + |E|)$ памяти. Такой список графически можно изобразить в виде таблицы, столбцов в которой – два, а строк не больше чем вершин в графе. В качестве примера возьмем смешанный граф (рис. 12.6).

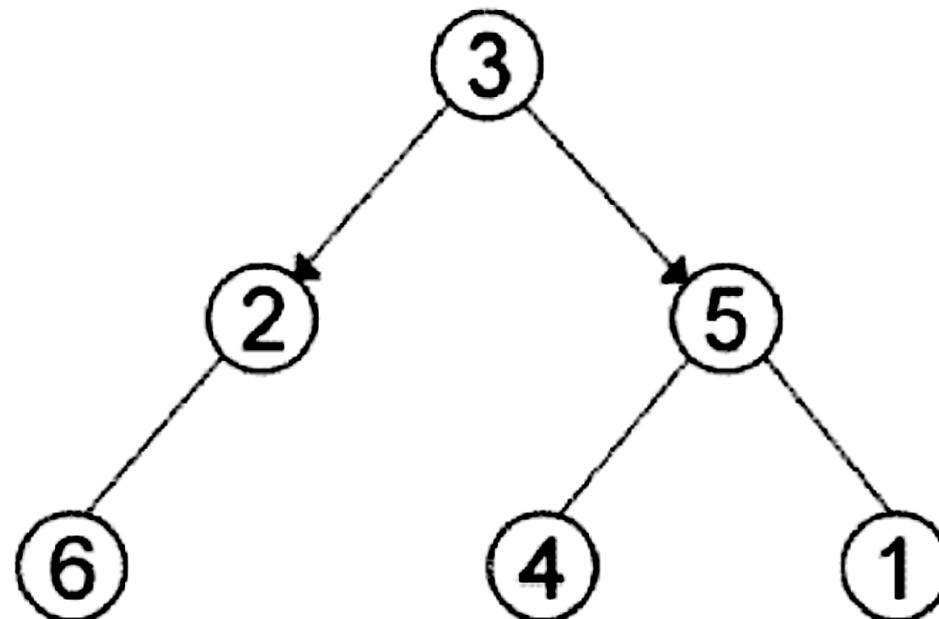


Рисунок 12.6

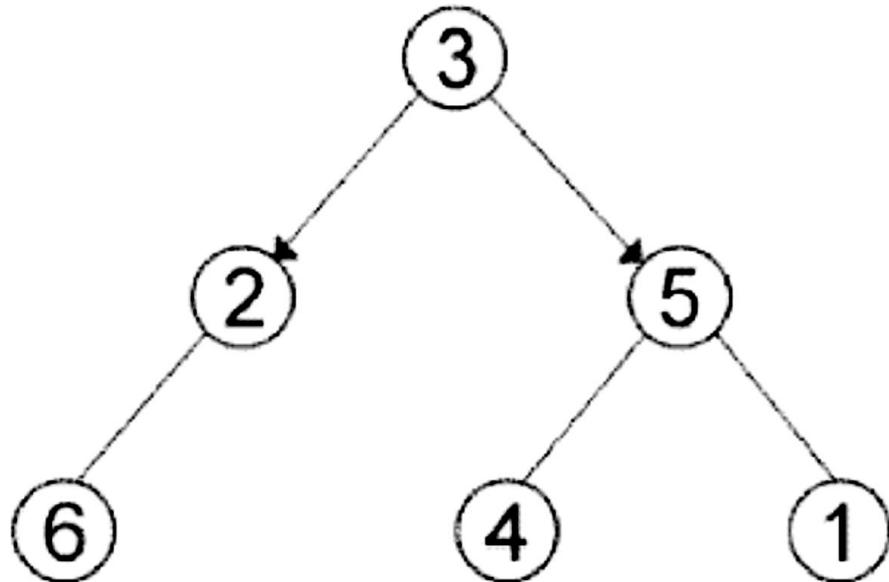


Рисунок 12.6

В нем 6 вершин ($|V|$) и 5 ребер ($|E|$). Из последних 2 ребра направленные и 3 ненаправленные, и, причем из каждой вершины выходит, как минимум одно ребро в другую вершину, из чего следует, что в списке смежности этого графа будет $|V|$ строк.

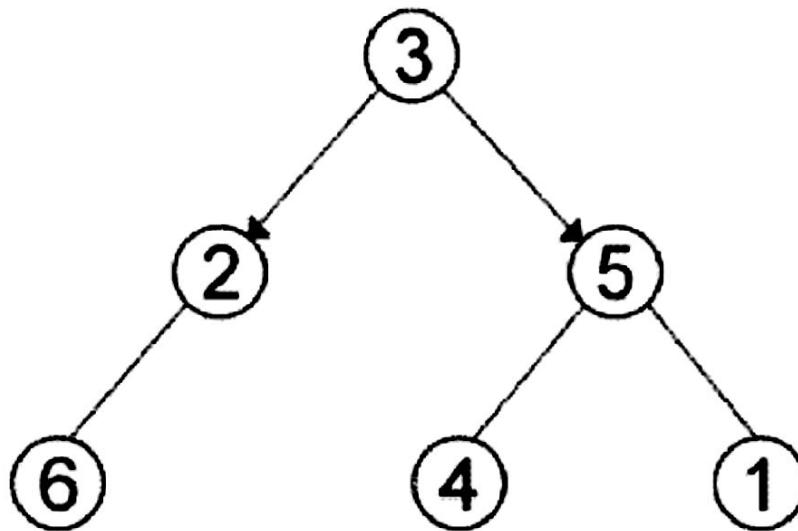


Рисунок 12.6

| Вершина выхода | Вершины входа |
|----------------|---------------|
| 1 | 5 |
| 2 | 6 |
| 3 | 2, 5 |
| 4 | 5 |
| 5 | 1, 4 |
| 6 | 2 |

В i строке и 1 столбце указана вершина выхода, а в i строке и 2 столбце – вершина входа. Так, к примеру, из вершины 5 выходят два ребра, входящие в вершины 1 и 4.

Теперь перейдем непосредственно к программной реализации списка смежности. Количество вершин и ребер будут задаваться с клавиатуры, поэтому установим ограничения, иначе говоря, определим две константы, одна из которых отвечает за максимально возможное число вершин (V_{max}), другая – ребер (E_{max}). Далее, нам понадобятся три одномерных целочисленных массива:

- $terminal[1..Emax]$ – хранит вершины, в которые входят ребра;
- $next[1..Emax]$ – содержит указатели на элементы массива $terminal$;
- $head[1..Vmax]$ – содержит указатели на начала подсписков, т. е. на такие вершины записанные в массив $terminal$, с которых начинается процесс перечисления всех вершин смежных одной i -ой вершине.

В программе позволительно выделить две основные части: ввод ребер с последующим добавлением их в список смежности, и вывод получившегося списка смежности на экран.

Код программы на C++:

```
#include <iostream>
using namespace std;
const int Vmax=100, Emax=Vmax*2;
int head[Vmax];
int next_el[Emax];
int terminal[Emax];
int n, m, i, j, k, v, u;
char r;
void Add(int v, int u) { //добавление ребра
    k=k+1;
    terminal[k]=u;
    next_el[k]=head[v];
    head[v]=k;
}
```

```
//главная функция
void main(){
    setlocale(LC_ALL, "Rus");
    k=0;
    cout<<"Кол. вершин >> "; cin>>n;
    cout<<"Кол. ребер >> "; cin>>m;
    cout<<"Вводите смежные вершины:"<<endl;
    for (i=0; i<m; i++){
        cin>>v>>u;
        cout<<"Ребро ориент.? (y/n) >> "; cin>>r;
        if (r=='y') Add(v, u);
```

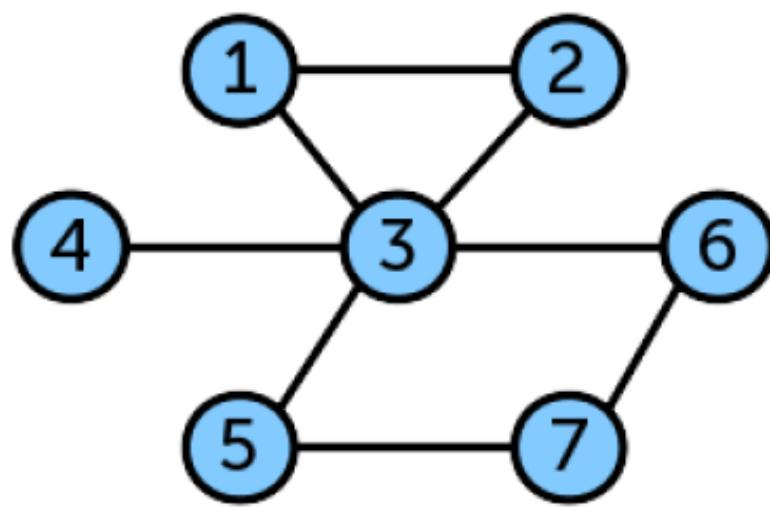
```
    else {
        Add(v, u);
        Add(u, v);
    }
    cout<<"..."<<endl;
}
//вывод списка смежности
cout<<"Список смежности графа:";
for (i=0; i<n+1; i++){
    j=head[i];
    if (i) cout<<i<<"->";
    while (j>0){
        if (!next_el[j]) cout<<terminal[j];
        else cout<<terminal[j]<<", ";
        j=next_el[j];
    }
    cout<<endl;
}
system("pause");
}
```

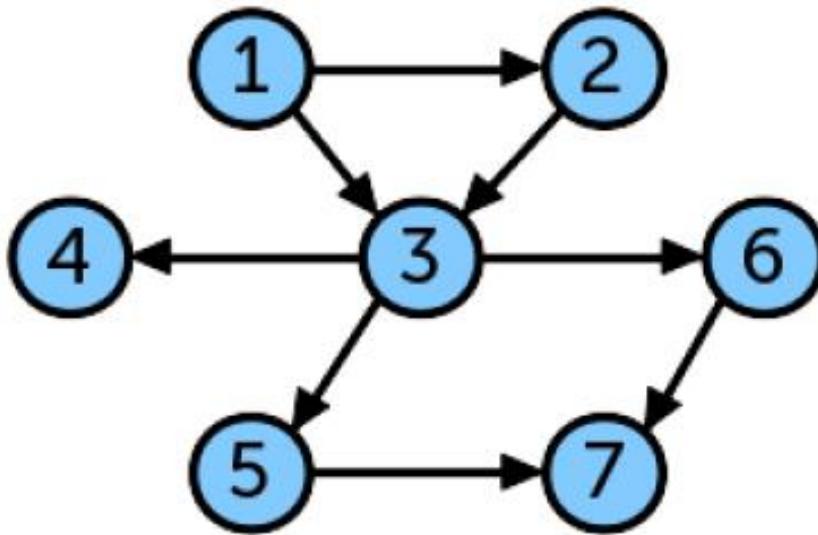
Первое действие, на которое стоит обратить внимание это запрос на ввод суммы вершин (n) и ребер (m) графа (пользователь должен заранее знать эти данные). Далее, запускается цикл ввода ребер (смежных вершин). Условие в этом цикле нужно для того, чтобы узнать, какое введено ребро. Если введено направленное ребро, то функция *add* вызывается 1 раз, иначе – 2, тем самым внося сведения, что есть ребро как из вершины v в вершину u , так и из u в v . После того как список смежности сформирован, программа выводит его на экран. Для этого использован цикл от 1 до n , где n – количество вершин, а также вложенный в него цикл, который прекращает свою работу тогда, когда указатель на следующий элемент для i -ой вершины отсутствует, т. е. все смежные ей вершины уже выведены.

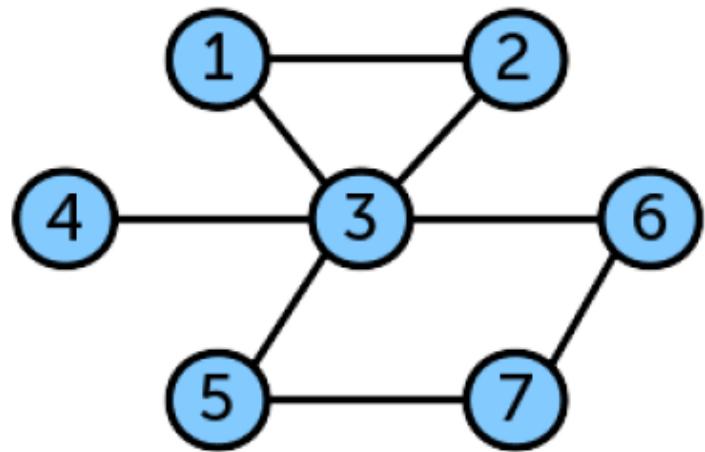
Функция *add* производит добавление ребер в изначально пустой список смежности. Чтобы сделать это, производятся операции с формальными параметрами, вспомогательной переменной k и тремя одномерными целочисленными массивами. Значение переменной k увеличивается на 1. Затем в k -ый элемент массива *terminal* записывается конечная для некоторого ребра вершина (*u*).

В третий строке k -ому элементу массива *next* присваивается адрес следующего элемента массива *terminal*. Ну и в конце массив *head* заполняется указателями на стартовые элементы, те с которых начинается подсписок (строка в таблице) смежных вершин с некоторой i -ой вершиной.

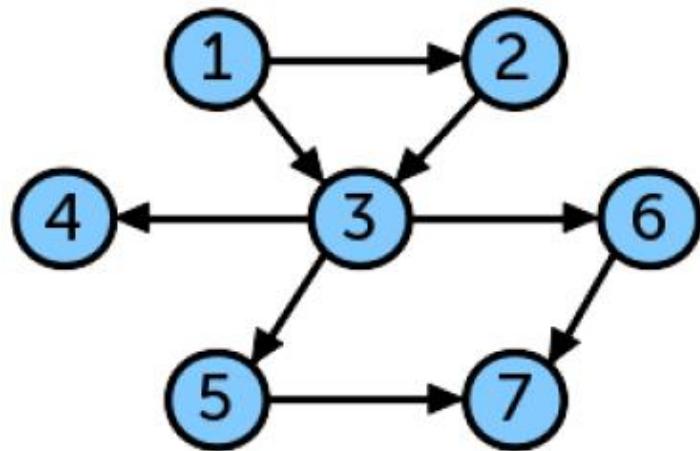
Так как в ячейке на пересечении i -ой строки и 2-ого столбца могут быть записаны несколько элементов (что соответствует нескольким смежным вершинам) назовем каждую строку в списке смежности его подсписком. Таким образом, в выведенном списке смежности, элементы подсписков будут отсортированы в обратном порядке. Но, как правило, порядок вывода смежных вершин (в подсписках) неважен.



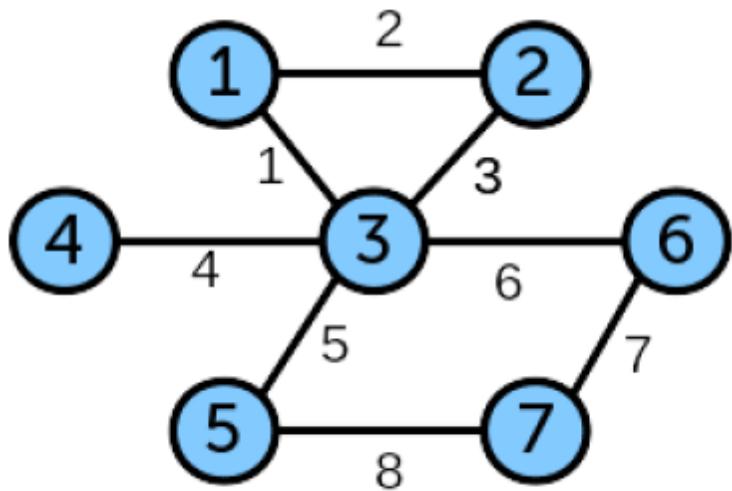




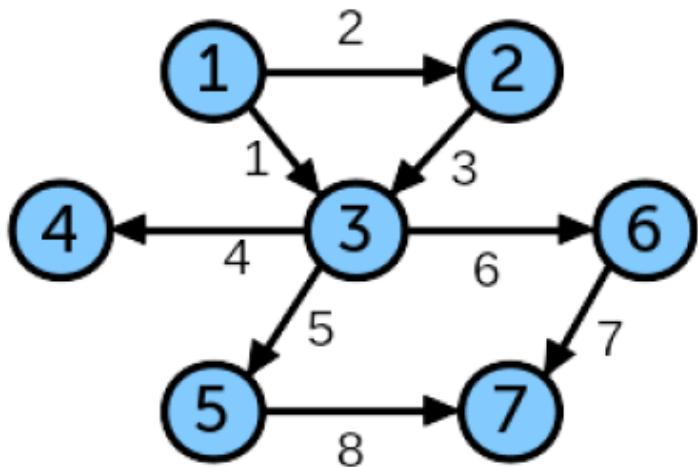
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



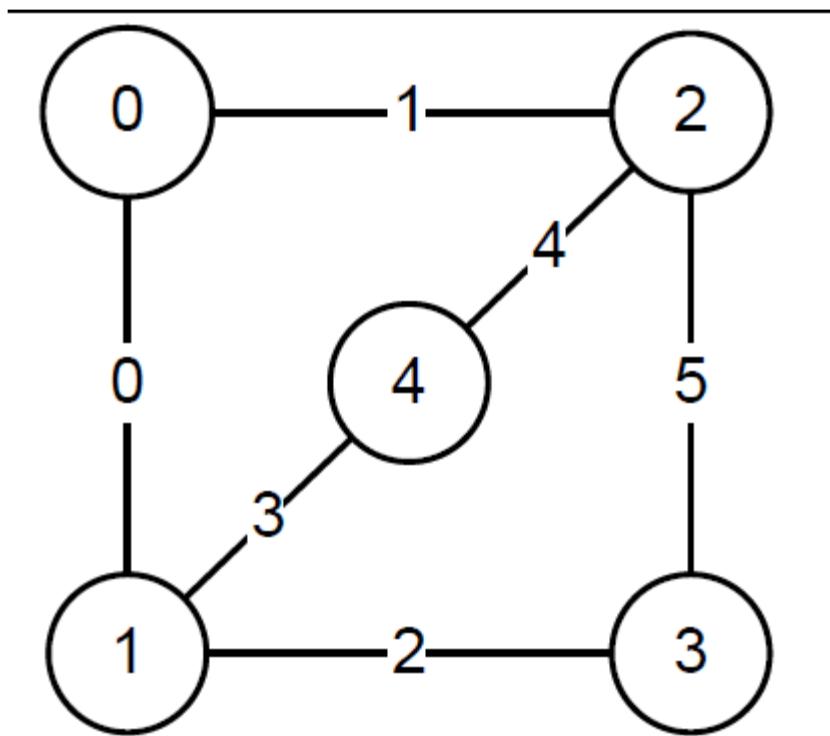
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

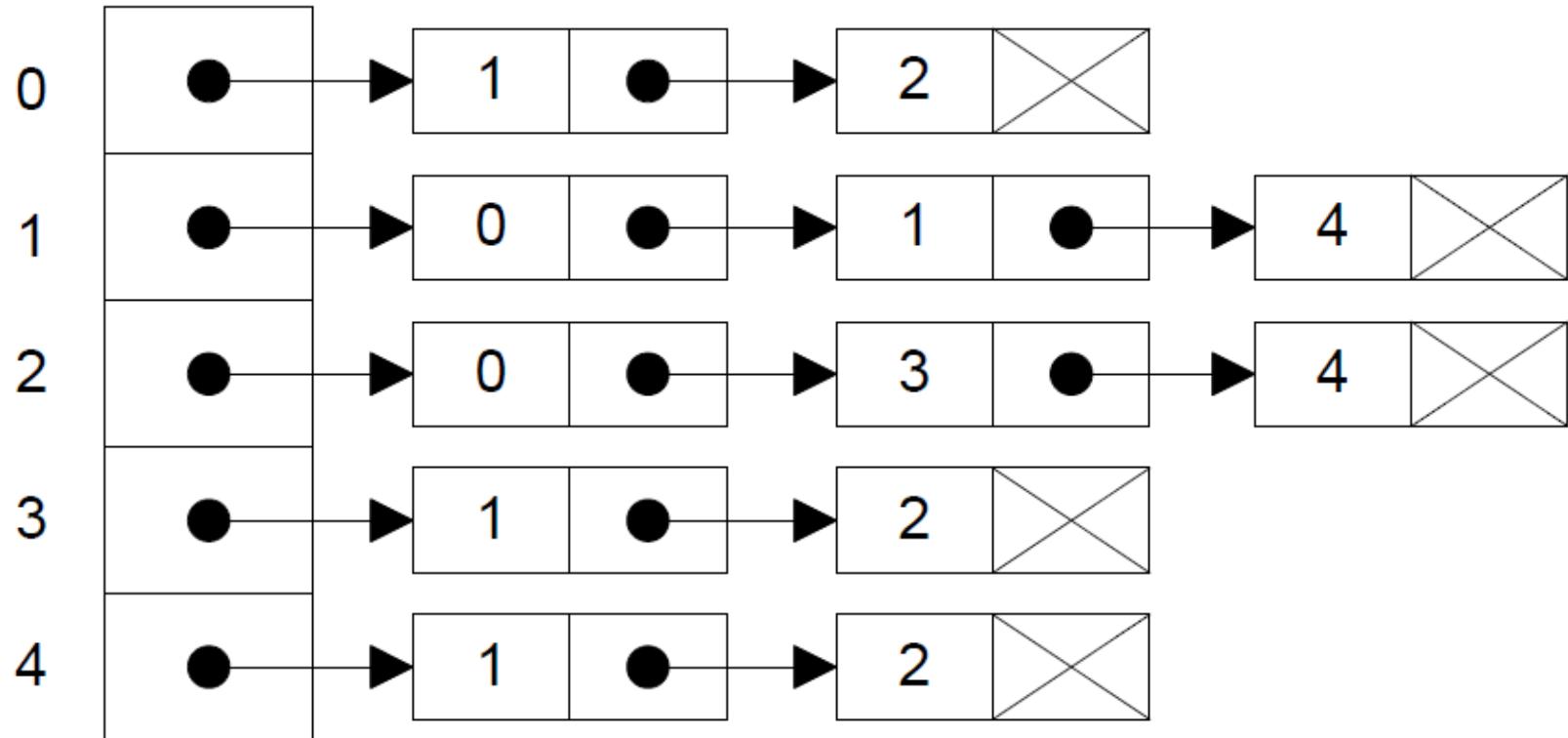


$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix}$$





$$AdjMat = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$IncMat = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Центральной частью поиска в глубину является рекурсивная операция `dfsVisit(u)`, которая посещает еще непосещенную вершину u . `dfsVisit(u)` записывает свою работу по посещению вершин с помощью окраски их в один из трех цветов.

Белый

Вершина еще не была посещена.

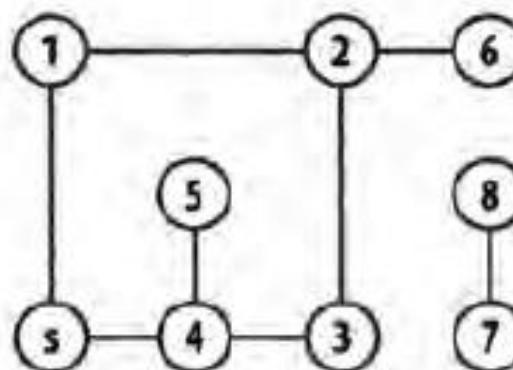
Серый

Вершина была посещена, но может иметь смежную вершину, которая еще не была посещена.

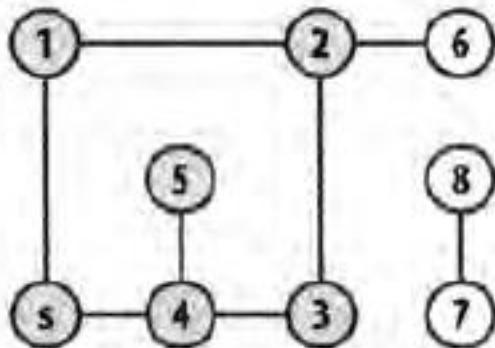
Черный

Вершина была посещена, как и все смежные с ней вершины.

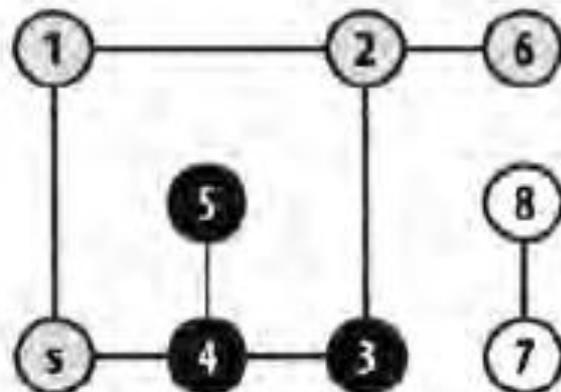
Изначально каждая вершина имеет белый цвет, указывающий, что она еще не была посещена, и поиск в глубину вызывает `dfsVisit` для исходной вершины s . `dfsVisit(u)` окрашивает вершину u в серый цвет перед тем, как рекурсивно вызвать `dfsVisit` для всех смежных с u и еще непосещенных вершин (т.е. окрашенных в белый цвет). После того как эти рекурсивные вызовы завершены, вершину u можно окрашивать в черный цвет, после чего осуществляется возврат из функции. После возврата из рекурсивной функции `dfsVisit` поиск в глубину откатывается к предыдущей вершине в поиске (на самом деле — к вершине, окрашенной в серый цвет), у которой имеются не посещенные соседние вершины, которые необходимо изучить. На рис. 6.7 показан пример работы алгоритма для небольшого графа.



Начиная с вершины s , `dfsVisit` рекурсивно посещает вершины (1–5), помечая каждую серым цветом, пока не будет найдена вершина без белых соседей (вершина 5)



По завершении каждого вызова `dfsVisit`
посещаются изначально пропущенные вершины
(например, вершина 6 является белым соседом вершины
2). по завершении работы вершина окрашивается
в черный цвет



Если граф несвязный,
в нем остается белая вершина
(она может быть не одна)

Контекст применения алгоритма

Поиск в глубину при обходе графа должен хранить только цвет каждой вершины (белый, серый или черный). Таким образом, для поиска в глубину необходимы накладные расходы памяти, равные $O(n)$.

Поиск в глубину может хранить свою информацию в массивах отдельно от графа. Поиску в глубину требуется только возможность обхода всех вершин графа, смежных с данной. Эта возможность позволяет легко выполнять поиск в глубину сложной информации, поскольку функция `dfsVisit` обращается к исходному графу за информацией о структуре с доступом только для чтения.

Анализ алгоритма

Рекурсивная функция `dfsVisit` вызывается один раз для каждой вершины графа. В `dfsVisit` необходимо проверить все соседние вершины; для ориентированных графов проход по ребрам осуществляется один раз, в то время как в неориентированных графах проход по ним осуществляется один раз, и еще один раз осуществляется их просмотр. В любом случае общая производительность алгоритма составляет $O(V + E)$.

Вариации алгоритма

Если исходный граф не связанный, то между s и некоторыми вершинами графа может не существовать пути; эти вершины будут оставаться непосещенными. Некоторые вариации гарантируют, что все вершины будут обработаны, достигая этого путем дополнительных выполнений `dfsVisit` для непосещенных вершин в методе `dfsSearch`. Если это будет сделано, то значения `pred[]` описывают лес поиска в глубину. Чтобы найти корни деревьев в этом лесу, следует сканировать массив `pred[]` на наличие в нем вершин r , для которых значения `pred[r]` равны -1 .

```
// Посещение вершины и графа и обновление информации
void dfsVisit(Graph const& graph, int u, /* вход */
              vector<int>&pred, vector<vertexColor>&color) /* выход */
{
    color[u] = Gray;

    // Обработка всех соседей u.
    for (VertexList::const_iterator ci = graph.begin(u);
         ci != graph.end(u); ++ci)
    {
        int v = ci->first;

        // Изучение непосещенных вершин и запись pred[].
        // По завершении вызова откат к смежным вершинам.
        if (color[v] == White)
        {
            pred[v] = u;
            dfsVisit(graph, v, pred, color);
        }
    }

    color[u] = Black; // Отмечаем, что все соседи пройдены.
}
```

```
/***
 * Выполнение поиска в глубину начиная с вершины s и вычисление
 * pred[u] - вершины, предшествующей u в лесу поиска в глубину.
 */
void dfsSearch(Graph const& graph, int s,      /* вход */
               vector<int>& pred)           /* выход */
{
    // Инициализация массива pred[]
    // и окраска всех вершин в белый цвет.
    const int n = graph.numVertices();
    vector<vertexColor> color(n, White);
    pred.assign(n, -1);
    // Поиск начиная с исходной вершины.
    dfsVisit(graph, s, pred, color);
}
```

Обход в глубину

Поиск в глубину (*Depth-first search*) – это рекурсивный алгоритм обхода вершин графа. Если при поиске в ширину график обходился симметрично, уровень за уровнем, то данный метод предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность дальнейшего продвижения, означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина). Отсутствие последнего свидетельствует об одной из двух возможных ситуаций: все вершины графа уже просмотрены, либо просмотрены все те, что доступны из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают этот вариант).

Рассмотрим то, как будет вести себя алгоритм на конкретном примере. Приведенный здесь неориентированный, связный граф имеет в сумме 5 вершин. Сначала необходимо выбрать начальную вершину. Какая бы вершина не была выбрана, граф в любом случае исследуется полностью, поскольку, как уже было сказано, это связный граф без единого направленного ребра. Пусть обход начнется с узла 1, тогда порядок последовательности просмотренных узлов будет следующим:

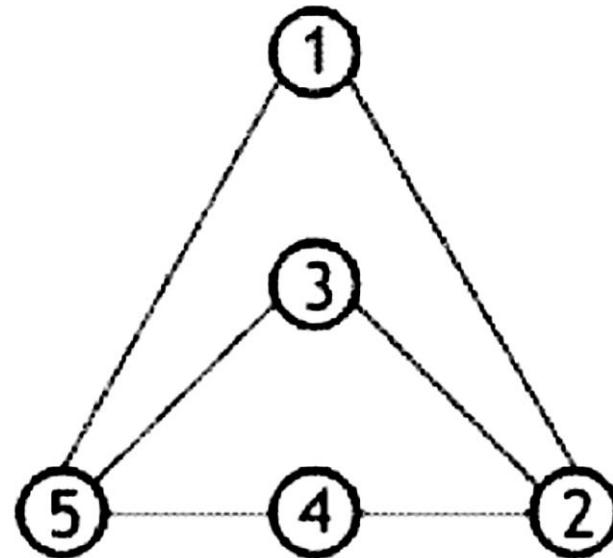


Рисунок 12.9

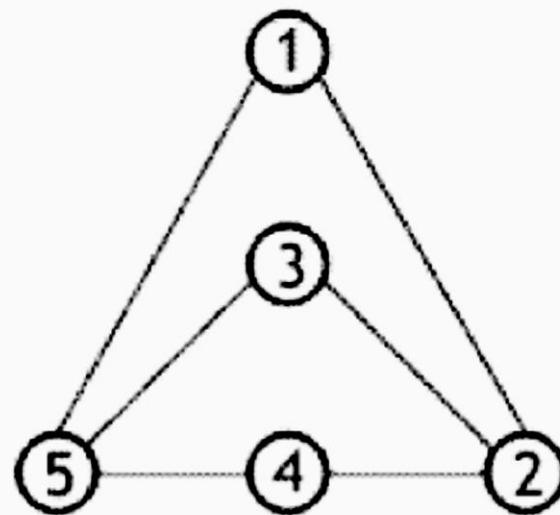


Рисунок 12.9

1 2 3 5 4

Если выполнение начать, например, с узла 3, то выходные данные станут такими:

3 2 1 5 4.

Алгоритм поиска в глубину реализуется при помощи рекурсии, т. е. участок программы, функция обхода по мере выполнения вызывает сама себя, что делает код в целом довольно коротким. Собственно вот псевдокод алгоритма:

- Заголовок функции $DFS(st)$;
- Вывести (st) ;
- $visited[st] \leftarrow$ посещена;
- Для $r=1$ до n выполнять;
- Если $(graph[st, r] \neq 0)$ и $(visited[r]$ не посещена) то $DFS(r)$.

Здесь ***DFS*** (*Depth-first search*) – название функции. Единственный ее параметр *st* – стартовый узел, передаваемый из главной части программы как аргумент. Каждому из элементов логического (булевого) массива *visited* заранее присваивается значение *false*, т. е. каждая из вершин изначально будет значиться как не посещённая. Последняя строка содержит двумерный массив *graph* – матрицу смежности графа. Собственно на этой строке стоит, пожалуй, сконцентрировать большую часть внимания. Функция завершается тогда, когда условие в данной строке станет ложно *n* раз. Если элемент матрицы смежности, на каком то шаге равен 1 (а не 0) и вершина с тем же номером, что и проверяемый столбец матрицы при этом не была посещена, то функция рекурсивно повторяется.

```
//Обход графа в глубину
#include <iostream>
using namespace std;
const int n=5;
int i, j;
bool *visited=new bool[n];
//матрица смежности графа
int graph[n][n] =
{
{0, 1, 0, 0, 1},
```

```
{1, 0, 1, 1, 0},
{0, 1, 0, 0, 1},
{0, 1, 0, 0, 1},
{1, 0, 1, 1, 0}
// _____
```

```
_____
void DFS(int st){
    int r;
    cout<<st+1<<" ";
    visited[st]=true;
    for (r=0; r<=n; r++)
        if ((graph[st][r]!=0) && (!visited[r]))
            DFS(r);
}
//_____
```

```
//Главная функция
void main(){
    setlocale(LC_ALL, "Rus");
    int start;
    cout<<"Матрица смежности графа: "<<endl;
    for (i=0; i<n; i++)
    {visited[i]=false;
    for (j=0; j<n; j++)
        cout<< " "<<graph[i][j];
    cout<<endl;
    }
    cout<<"Стартовая вершина >> ";
    cin>>start;
    //массив посещенных вершин
    bool *vis=new bool[n];
    cout<<"Порядок обхода: ";
    DFS(start-1);
    delete []visited;
    system("pause>>void");
}
```

Поиск в ширину представляет собой другой подход, отличный от поиска в глубину. При поиске в ширину систематически посещаются все вершины графа $G = (V, E)$, которые находятся на расстоянии k ребер от исходной вершины s , перед посещением любой вершины на расстоянии $k + 1$ ребер. Этот процесс повторяется до тех пор, пока не останется вершин, достижимых из s . Этот поиск не посещает вершины графа G , которые не достижимы из s . Алгоритм работает как для неориентированных, так и для ориентированных графов.

Поиск в ширину гарантированно находит кратчайший путь в графе от вершины s до целевой вершины, хотя в процессе работы может выполнить вычисления для весьма большого количества узлов. Поиск в глубину пытается найти путь как можно быстрее, но цена этой скорости — отсутствие гарантии того, что этот путь будет кратчайшим.

Поиск в ширину работает без необходимости какого-либо возврата. Он также окрашивает вершины в белый, серый или черный цвет, как и поиск в глубину. Смысл этих цветов в точности тот же, что и при поиске в глубину. Для сравнения с поиском в глубину рассмотрим поиск в ширину на том же графе (рис. 6.6) в момент, когда, как и ранее, в черный цвет окрашена пятая вершина (вершина 2), как показано на рис. 6.10. В момент, показанный на рисунке, поиск в ширину окрасил в черный цвет исходную вершину s , вершины, отстоящие от нее на одно ребро — {1, 6 и 8}, — и вершину 2, отстоящую на два ребра от s .

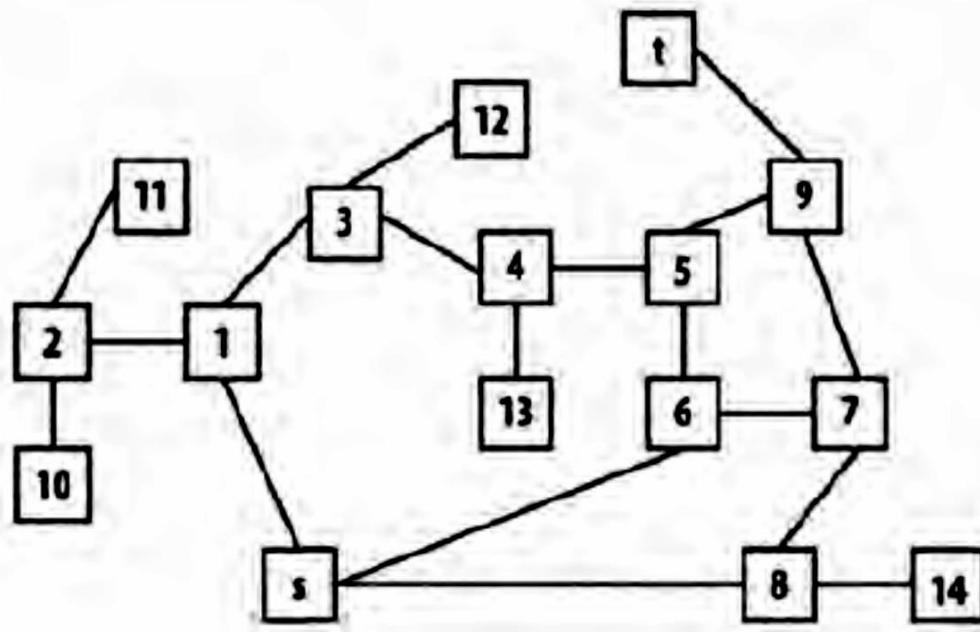


Рис. 6.6. Представление лабиринта на рис. 6.5 в виде графа

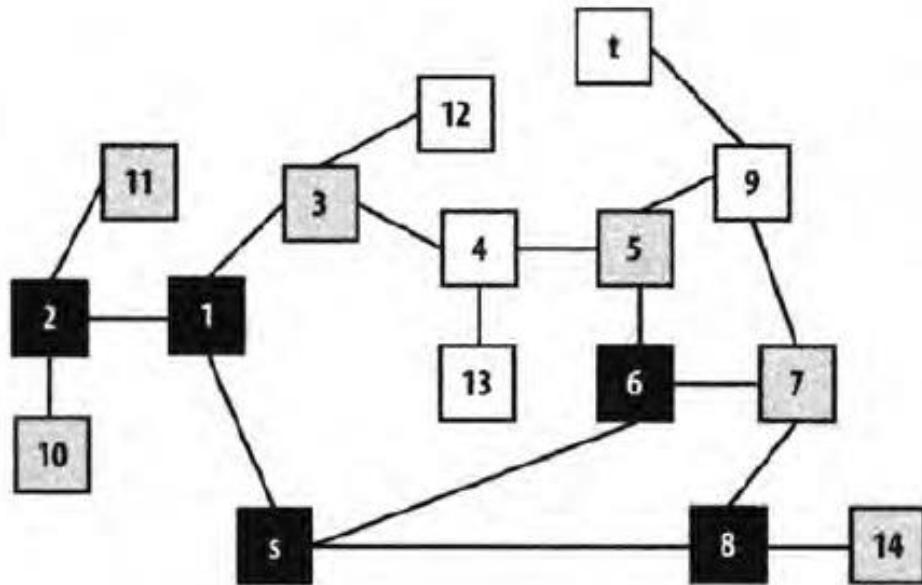
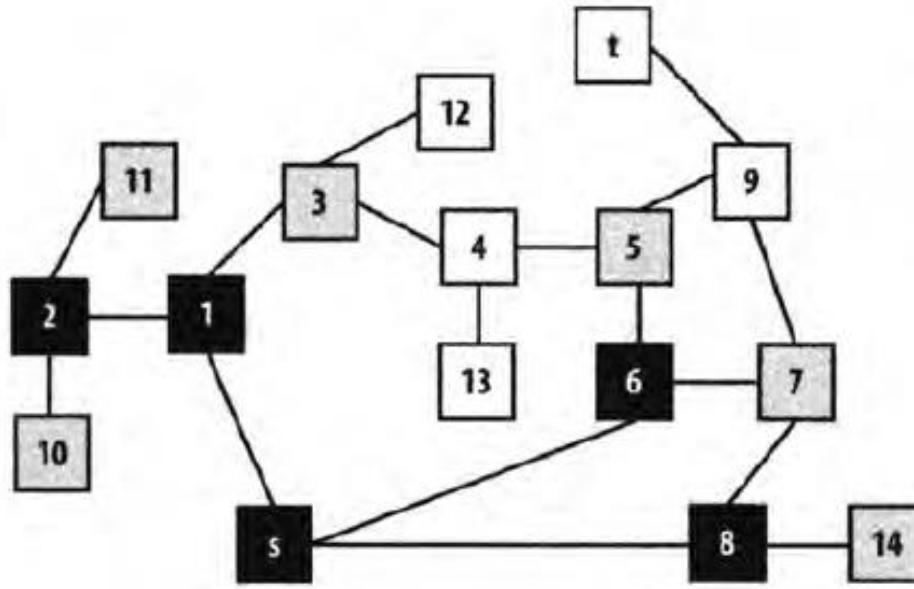


Рис. 6.10. Поиск в ширину на графике с рис. 6.7 после окраски в черный цвет пяти вершин

| v | pred | dist |
|----|------|------|
| s | -1 | 0 |
| 1 | s | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 2 |
| 4 | * | * |
| 5 | 6 | 2 |
| 6 | s | 1 |
| 7 | 6 | 2 |
| 8 | s | 1 |
| 9 | * | * |
| 10 | 2 | 3 |
| 11 | 2 | 3 |
| 12 | * | * |
| 13 | * | * |
| 14 | 8 | 2 |
| t | * | * |



Все остальные вершины на расстоянии двух ребер от s — $\{3, 5, 7, 14\}$ — находятся в очереди Q ожидающих обработки. Там же есть и две вершины, отстоящие от исходной вершины s на три ребра — $\{10, 11\}$. Обратите внимание, что все вершины, находящиеся в очереди, окрашены в серый цвет, отражающий их активное состояние.

Пример 6.2. Реализация поиска в ширину

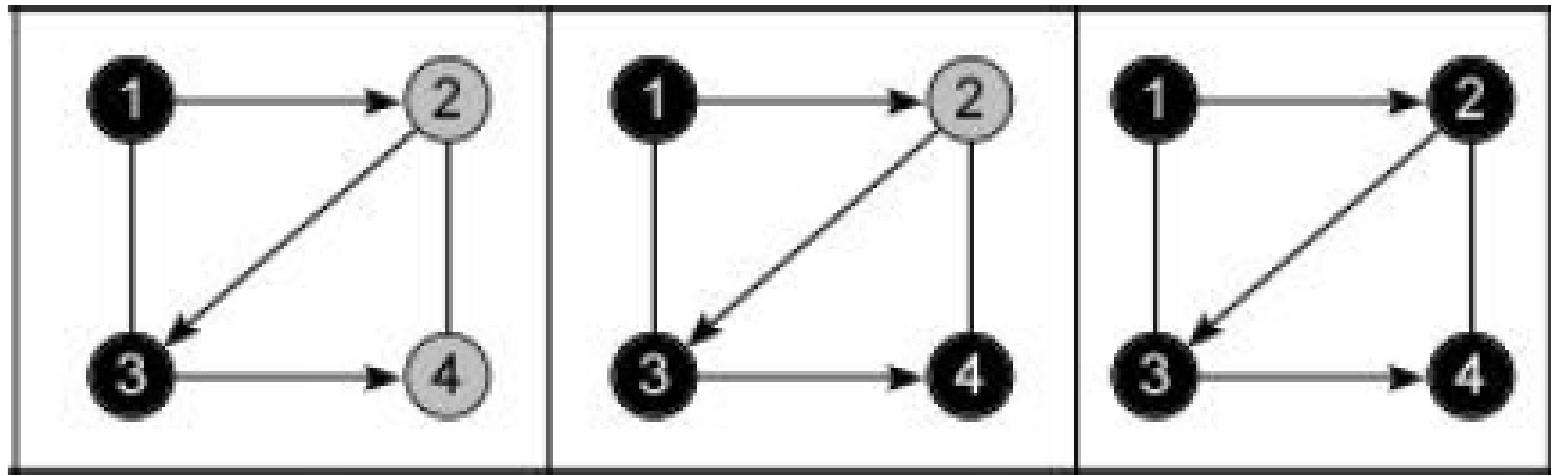
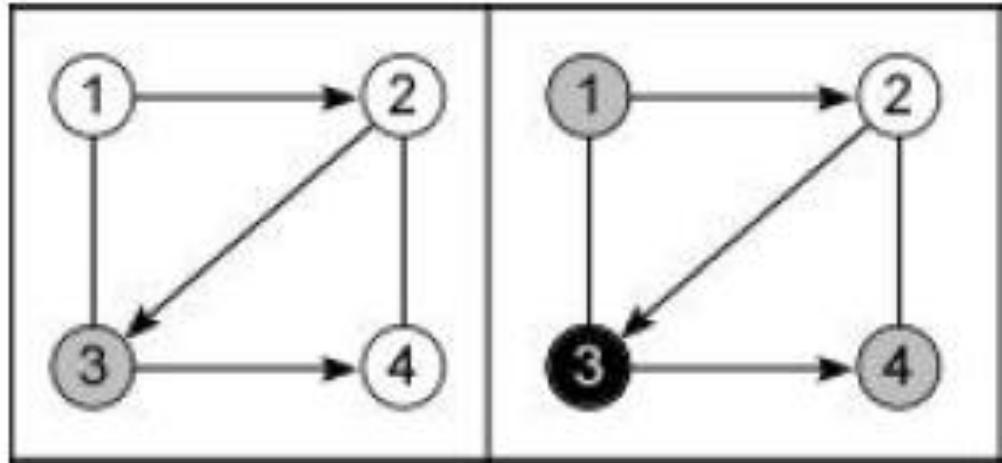
```
/**  
 * Выполняет поиск в ширину в графе из вершины s и вычисляет  
 * BFS-расстояние и предыдущую вершину для всех вершин графа.  
 */  
void bfsSearch(Graph const &graph, int s,           /* Вход */  
                vector<int>&dist, vector<int>&pred) /* Выход */  
{  
    // Инициализация dist и pred для пометки непосещенных вершин.  
    // Начинаем с s и окрашиваем ее в серый цвет, так как ее  
    // соседи еще не посещены.  
    const int n = graph.numVertices();  
    pred.assign(n, -1);  
    dist.assign(n, numeric_limits<int>::max());  
    vector<vertexColor> color (n, White);  
  
    dist[s] = 0;  
    color[s] = Gray;  
  
    queue<int> q;  
    q.push(s);  
    while (!q.empty()) {  
        int u = q.front();  
  
        // Исследуем соседей u для расширения горизонта поиска
```

```
for(VertexList::const_iterator ci = graph.begin (u);
    ci != graph.end (u); ++ci) {
    int v = ci->first;
    if (color[v] == White) {
        dist[v] = dist[u]+1;
        pred[v] = u;
        color[v] = Gray;
        q.push(v);
    }
}

q.pop();
color[u] = Black;
}
}
```

Обход при помощи поиска в ширину, подразумевает прохождение графа, начиная с некоторой вершины p , вширь, т. е. следующим шагом, после посещения вершины p , будет посещение смежных с ней вершин (обозначим как $q \subseteq V$, где V – множество всех вершин, а q – некоторое его подмножество). Далее эта процедура повториться для вершин смежных с вершинами из подмножества q , кроме вершины p , т. к. она уже была просмотрена. Продолжая действовать по такому принципу, алгоритм обойдет все оставшиеся вершины из множества V . После просмотра всех вершин алгоритм прекращает свою работу.

Поиск в ширину можно рассматривать как последовательное прохождение всех уровней графа начиная с произвольной вершины. На показанном рисунке в качестве нее взята вершина 3. Сначала она помечается серым, как обнаруженная, а затем черным, поскольку обнаружены смежные с ней вершины (1 и 4), которые, в свою очередь, помечаются серым. Находятся «соседи» вершины 1 (2) и закрашиваются серым, а она, соответственно, черным. И наконец, узлы 4 и 2, в данном порядке, просматриваются, обход в ширину завершается.



Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах, дать понять это был призван смешанный граф, используемый в качестве примера. Стоить отметить, что в неориентированном связном графе данный метод сможет обойти все имеющиеся узлы, а в смешанном и орграфе это необязательно произойдет. К тому же, несмотря на то, что до сих пор рассматривался обход всех вершин, вполне вероятно, достаточным окажется, например просмотр определенного их количества или нахождение конкретной вершины. В таком случае нужно немного подкорректировать алгоритм, а не изменять его полностью или вовсе отказаться от такового.

Теперь перейдем к более формальному и подробному описанию поиска в ширину. Для начала укажем два основных объекта, на чем строится весь алгоритм: очередь (*queue*) и массив посещенных вершин (*vis*). Они оба выполнены в виде массива, первый – целочисленного (в зависимости от типа узлов), второй – логического. Уже посещенные вершины, заносятся в массив *vis*, что предотвратит зацикливание, а *queue* будет хранить задействованные узлы. Напомним, что структура данных «очередь» работает по принципу «первый пришел – первый вышел». Рассмотрим процесс полного обхода графа поэтапно:

1. массив vis обнуляется, т. е. ни одна вершина графа еще не была посещена;
2. в качестве стартовой, выбирается некоторая вершина r и помещается в очередь (в массив $queue$);
3. вершина r исследуется (помечается как посещенная), и все смежные с ней вершины помещаются в конец очереди, а сама она удаляется;
4. если на данном этапе очередь оказывается пустой, то алгоритм останавливается;
5. посещается вершина u , стоящая в начале очереди, помечается как посещенная, и все ее потомки заносятся в конец очереди;
6. процесс продолжается, начиная с пункта 4.

Поиск в ширину, начиная со стартовой вершины, постепенно уходит от нее все дальше и дальше, спускаясь, уровень за уровнем вниз. Получается, что по окончанию работы алгоритма будут найдены все кратчайшие пути из начальной вершины до каждого из узлов графа. Но при этом близость определяется количеством ребер, лежащих между двумя точками, т. е. если график взвешенный, то выгода необязательно тождественна краткости.

```
//Обход графа в ширину
#include <iostream>
using namespace std;
const int n=4;
int i, j;
//матрица смежности графа
int GM[n][n] =
{
{0, 1, 1, 0},
{0, 0, 1, 1},
{1, 0, 0, 1},
{0, 1, 0, 0}
};
//
```

```
void BFS(bool *passed, int unit) { //очередь
    int *queue=new int[n]; //указатели очереди
    int count, head;
    for (i=0; i<n; i++) queue[i]=0;
    count=0; head=0;
    queue[count++]=unit;
    passed[unit]=true;
    while (head<count){
        unit=queue[head++];
        cout<<unit+1<<" ";
        for (i=0; i<n; i++)
            if (GM[unit][i] && !passed[i]) {
                queue[count++]=i;
                passed[i]=true;
            }
    }
    delete []queue;
}
//
```

```
//Главная функция
void main(){
    setlocale(LC_ALL, "Rus");
    int start;
    cout<<"Стартовая вершина >> ";
    cin>>start;
    //массив посещенных вершин
    bool *vis=new bool[n];
    cout<<"Матрица смежности графа: "<<endl;
    for (i=0; i<n; i++){
        vis[i]=false;
        for (j=0; j<n; j++) cout<<" "<<GM[i][j];
        cout<<endl;
    }
    cout<<"Порядок обхода: ";
    BFS(vis, start-1);
    cout<<endl;
    delete []vis;
    system("pause");
}
```

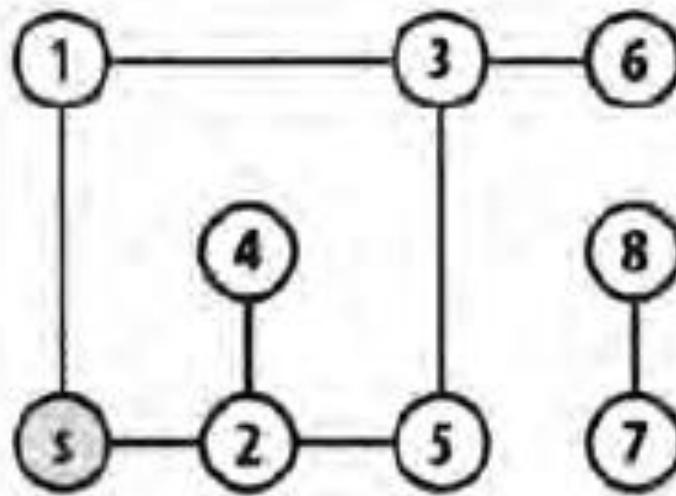
Поиск в ширину представляет собой другой подход, отличный от поиска в глубину. При поиске в ширину систематически посещаются все вершины графа $G = (V, E)$, которые находятся на расстоянии k ребер от исходной вершины s , перед посещением любой вершины на расстоянии $k + 1$ ребер. Этот процесс повторяется до тех пор, пока не останется вершин, достижимых из s . Этот поиск не посещает вершины графа G , которые не достижимы из s . Алгоритм работает как для неориентированных, так и для ориентированных графов.

Поиск в ширину гарантированно находит кратчайший путь в графе от вершины s до целевой вершины, хотя в процессе работы может выполнить вычисления для весьма большого количества узлов. Поиск в глубину пытается найти путь как можно быстрее, но цена этой скорости — отсутствие гарантии того, что этот путь будет кратчайшим.

Поиск в ширину работает без необходимости какого-либо возврата. Он также окрашивает вершины в белый, серый или черный цвет, как и поиск в глубину. Смысл этих цветов в точности тот же, что и при поиске в глубину. Для сравнения с поиском в глубину рассмотрим поиск в ширину на том же графе (рис. 6.6) в момент, когда, как и ранее, в черный цвет окрашена пятая вершина (вершина 2), как показано на рис. 6.10. В момент, показанный на рисунке, поиск в ширину окрасил в черный цвет исходную вершину s , вершины, отстоящие от нее на одно ребро — {1, 6 и 8}, — и вершину 2, отстоящую на два ребра от s .

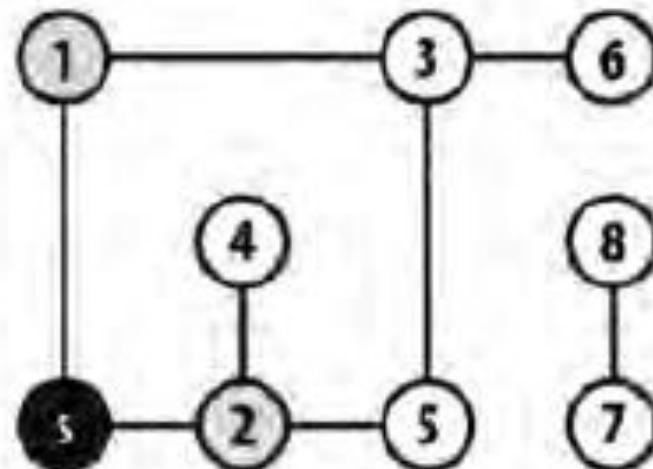
Изначально все вершины,
кроме s , белые

$$Q = \boxed{s}$$



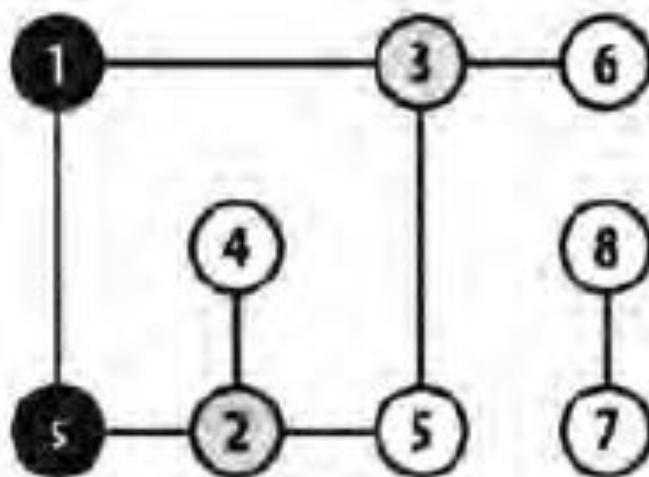
После первой
итерации цикла

$$Q = \begin{bmatrix} 1 & 2 \end{bmatrix}$$



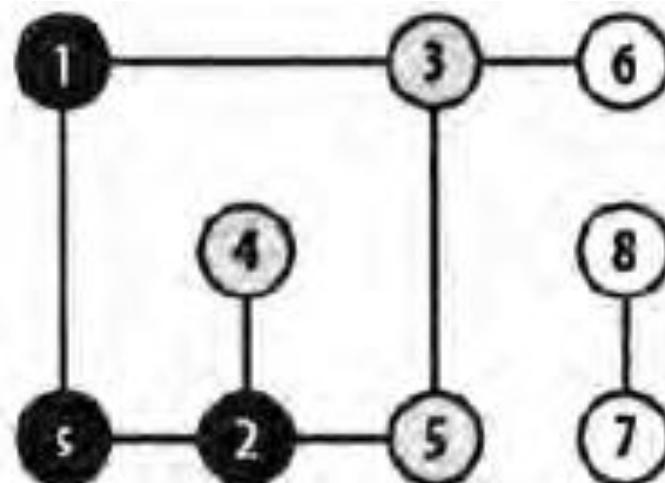
После второй
итерации цикла

$$Q = \begin{bmatrix} 2 & 3 \end{bmatrix}$$



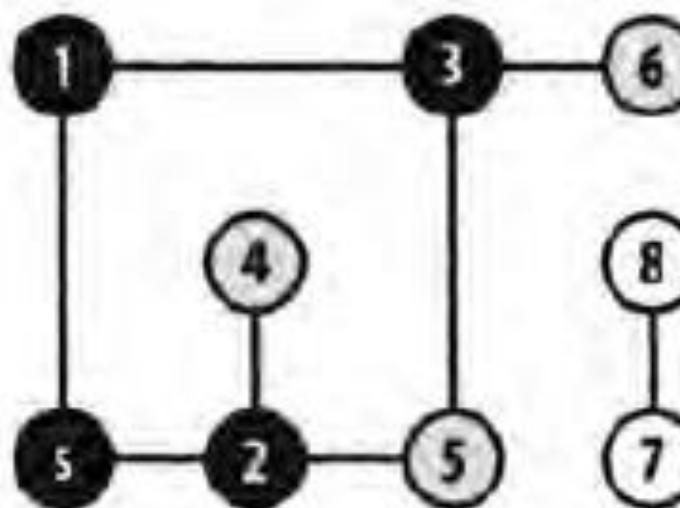
После третьей
итерации цикла

$$Q = \boxed{3 \quad 4 \quad 5}$$



После четвертой
итерации цикла

$$Q = \boxed{4 \quad 5 \quad 6}$$



Алгоритм Дейкстры

Алгоритм голландского ученого Эдсгера Дейкстры находит все кратчайшие пути из одной изначально заданной вершины графа до всех остальных. С его помощью, при наличии всей необходимой информации, можно, например, узнать какую последовательность дорог лучше использовать, чтобы добраться из одного города до каждого из многих других, или в какие страны выгодней экспортировать нефть и т. п. Минусом данного метода является невозможность обработки графов, в которых имеются ребра с отрицательным весом, т. е. если, например, некоторая система предусматривает убыточные для Вашей фирмы маршруты, то для работы с ней следует воспользоваться отличным от алгоритма Дейкстры методом.

Для реализации алгоритма понадобиться два массива: логический *visited* для хранения информации о посещенных вершинах и численный *distance*, в который будут заноситься найденные кратчайшие пути. Итак, имеется граф $G=(V, E)$. Каждая из вершин входящих во множество V , изначально отмечена как не посещенная, т. е. элементам массива *visited* присвоено значение *false*.

Поскольку самые выгодные пути только предстоит найти, в каждый элемент вектора *distance* записывается такое число, которое заведомо больше любого потенциального пути (обычно это число называют бесконечностью, но в программе используют, например максимальное значение конкретного типа данных). В качестве исходного пункта выбирается вершина s и ей приписывается нулевой путь: $distance[s] \leftarrow 0$, т. к. нет ребра из s в s (метод не предусматривает петель). Далее, находятся все соседние вершины (в которые есть ребро из s) [пусть таковыми будут t и u] и поочередно исследуются, а именно вычисляется стоимость маршрута из s поочередно в каждую из них:

- $distance[t] = distance[s] + \text{вес инцидентного } s \text{ и } t \text{ ребра};$
- $distance[u] = distance[s] + \text{вес инцидентного } s \text{ и } u \text{ ребра}.$

Но вполне вероятно, что в ту или иную вершину из s существует несколько путей, поэтому цену пути в такую вершину в массиве $distance$ придется пересматривать, тогда наибольшее (неоптимальное) значение игнорируется, а наименьшее ставиться в соответствие вершине.

После обработки смежных с s вершин она помечается как посещенная: $\text{visited}[s] \leftarrow \text{true}$, и активной становится та вершина, путь из s в которую минимален. Допустим, путь из s в u короче, чем из s в t , следовательно, вершина u становится активной и выше описанным образом исследуются ее соседи, за исключением вершины s . Далее u помечается как пройденная: $\text{visited}[u] \leftarrow \text{true}$, активной становится вершина t и вся процедура повторяется для нее. Алгоритм Дейкстры продолжается до тех пор, пока все доступные из s вершины не будут исследованы.

Теперь на конкретном графе проследим работу алгоритма, найдем все кратчайшие пути между истоковой и всеми остальными вершинами (рис. 12.10). Размер (количество ребер) изображенного ниже графа равен 7 ($|E|=7$), а порядок (количество вершин) – 6 ($|V|=6$). Это взвешенный граф, каждому из его ребер поставлено в соответствие некоторое числовое значение, поэтому ценность маршрута необязательно определяется числом ребер, лежащих между парой вершин.

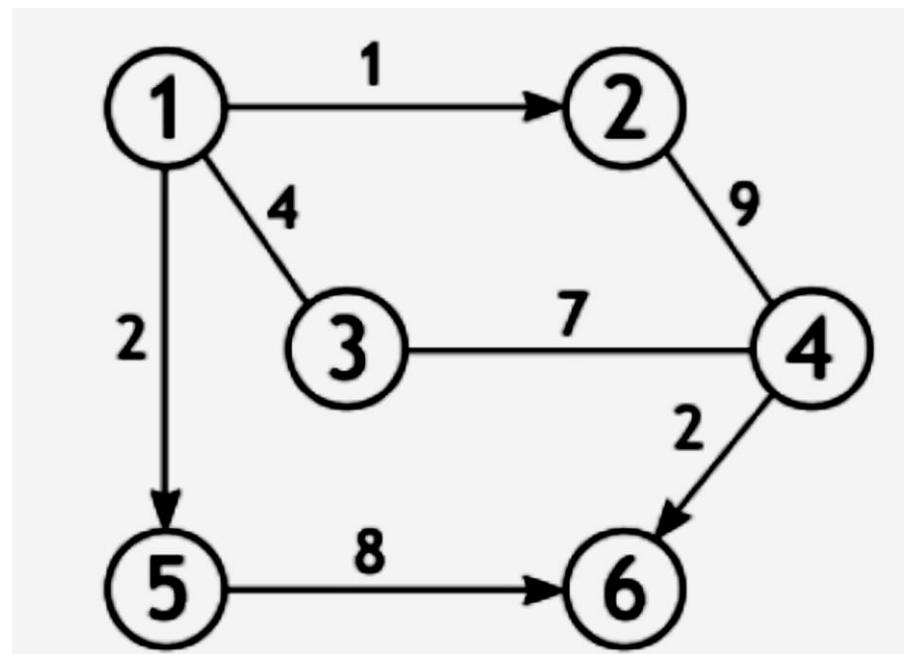


Рисунок 12.10

Из всех вершин входящих во множество V выберем одну, ту, от которой необходимо найти кратчайшие пути до остальных доступных вершин. Пусть таковой будет вершина 1. Длина пути до всех вершин, кроме первой, изначально равна бесконечности, а до нее – 0, т. к. граф не имеет петель (рис. 12.11).

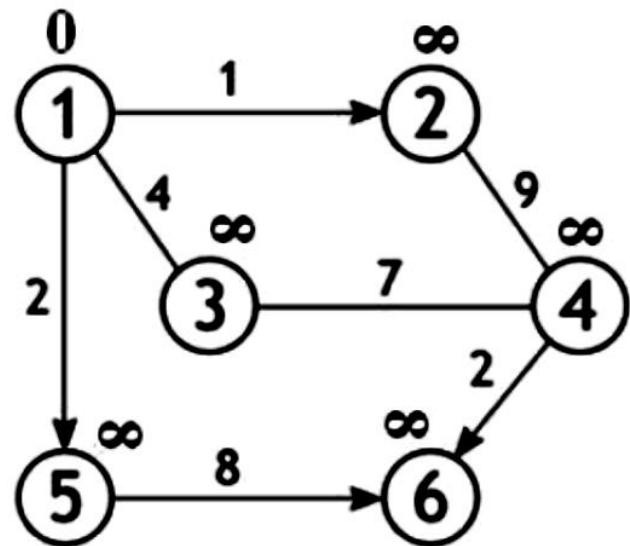


Рисунок 12.11

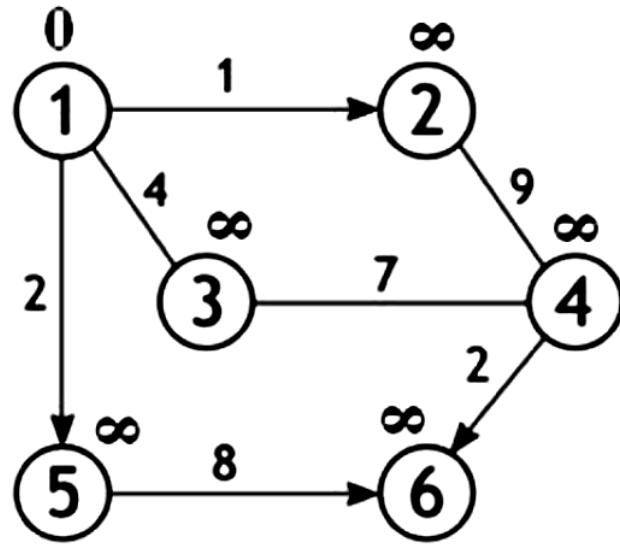


Рисунок 12.11

У вершины 1 ровно 3 соседа (вершины 2, 3, 5), и чтобы вычислить длину пути до них нужно сложить вес дуг, лежащих между вершинами: 1 и 2, 1 и 3, 1 и 5 со значением первой вершины (с нулем):

- $2 \leftarrow 1 + 0$
- $3 \leftarrow 4 + 0$
- $5 \leftarrow 2 + 0$

Как уже отмечалось, получившиеся значения присваиваются вершинам, лишь в том случае если они «лучше» (меньше) тех которые значатся на настоящий момент. А так как каждое из трех чисел меньше бесконечности, они становятся новыми величинами, определяющими длину пути из вершины 1 до вершин 2, 3 и 5 (рис. 12.12).

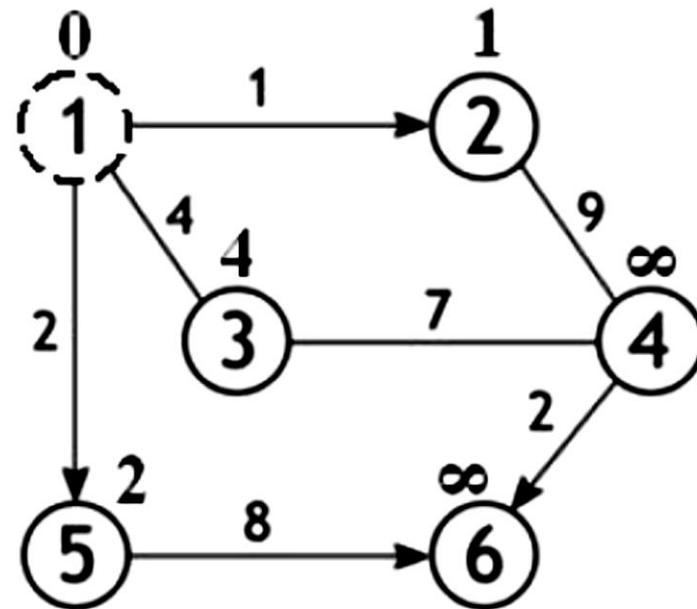


Рисунок 12.12

Далее, активная вершина помечается как посещенная, статус «активной» (круг, обозначенный пунктиром) переходит к одной из ее соседок, а именно к вершине 2, поскольку она ближайшая к ранее активной вершине (рис. 12.13).

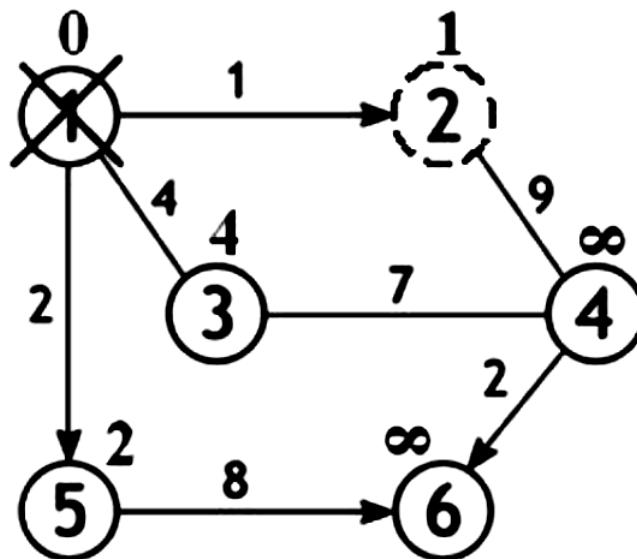


Рисунок 12.13

У вершины 2 всего один не рассмотренный сосед (вершина 1 помечена как посещенная), расстояние до которого из нее равно 9, но нам необходимо вычислить длину пути из истоковой вершины, для чего нужно сложить величину приписанную вершине 2 с весом дуги из нее в вершину 4:

$$- 4 \leftarrow 1 + 9$$

Условие «краткости» ($10 < \infty$) выполняется, следовательно, вершина 4 получает новое значение длины пути (рис. 12.14).

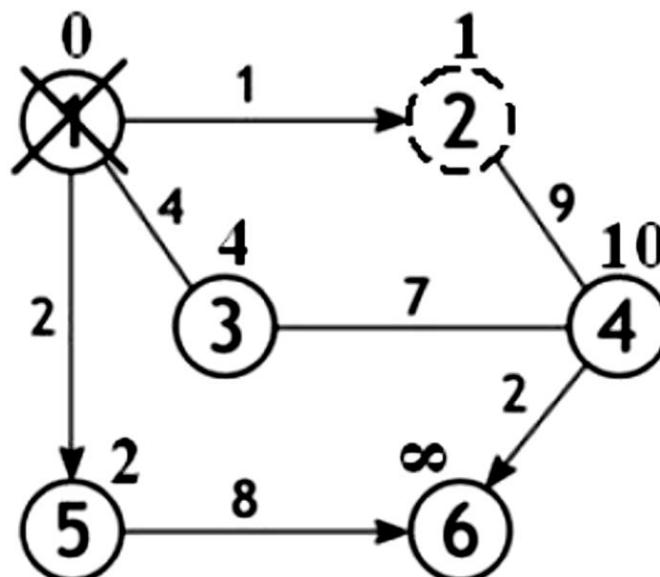


Рисунок 12.14

Вершина 2 перестает быть активной, также как и вершина 1 удаляется из списка не посещённых. Теперь тем же способом исследуются соседи вершины 5, и вычисляется расстояние до них (рис. 12.15).

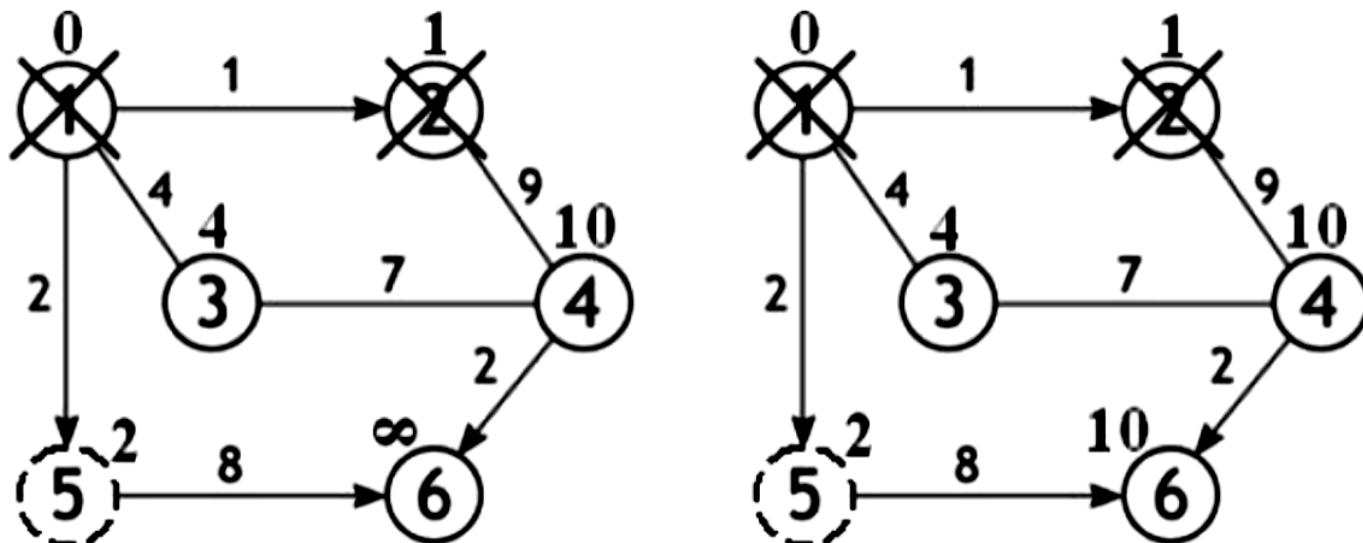


Рисунок 12.15

Когда дело доходит до осмотра соседей вершины 3, то тут важно не ошибиться, т. к. вершина 4 уже была исследована и расстояние одного из возможных путей из истока до нее вычислено. Если двигаться в нее через вершину 3, то путь составит $4+7=11$, а $11 > 10$, поэтому новое значение игнорируется, старое остается (рис. 12.16).

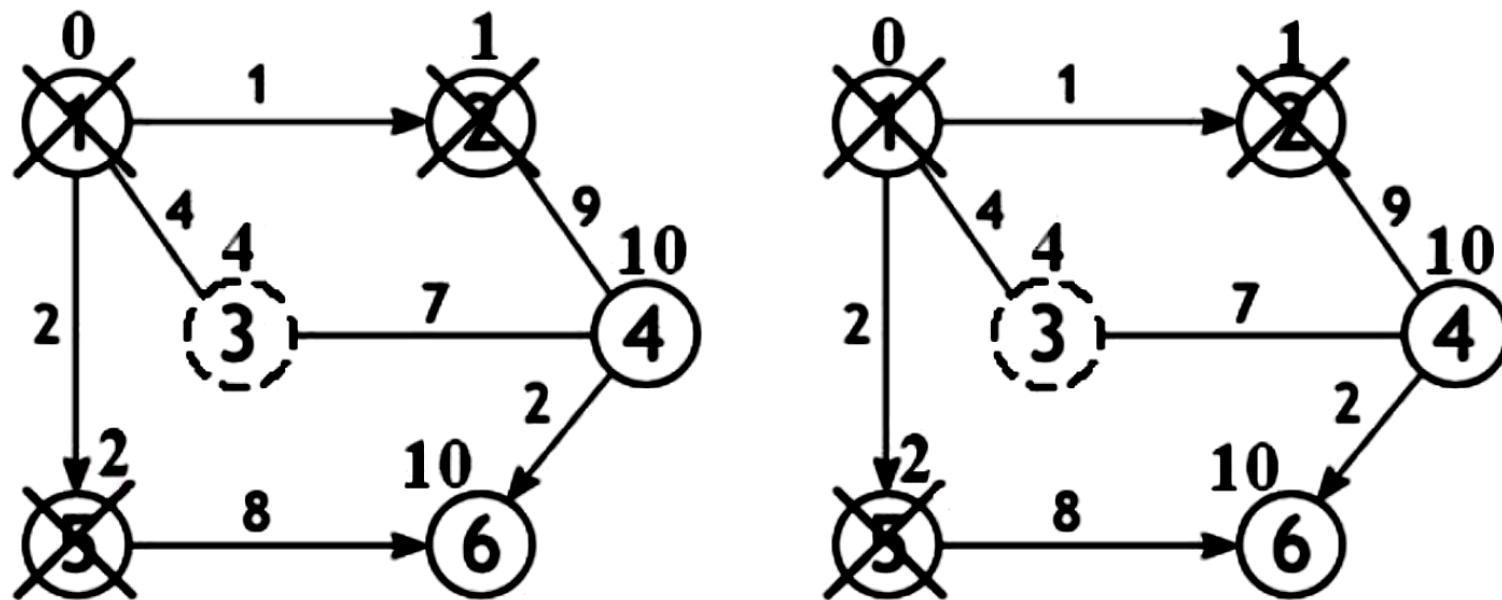


Рисунок 12.16

Та же ситуация выходит и с вершиной 6. Значение самого близкого пути до нее из вершины 1 равно 10, а оно получается только в том случае если идти через вершину 5 (рис. 12.17).

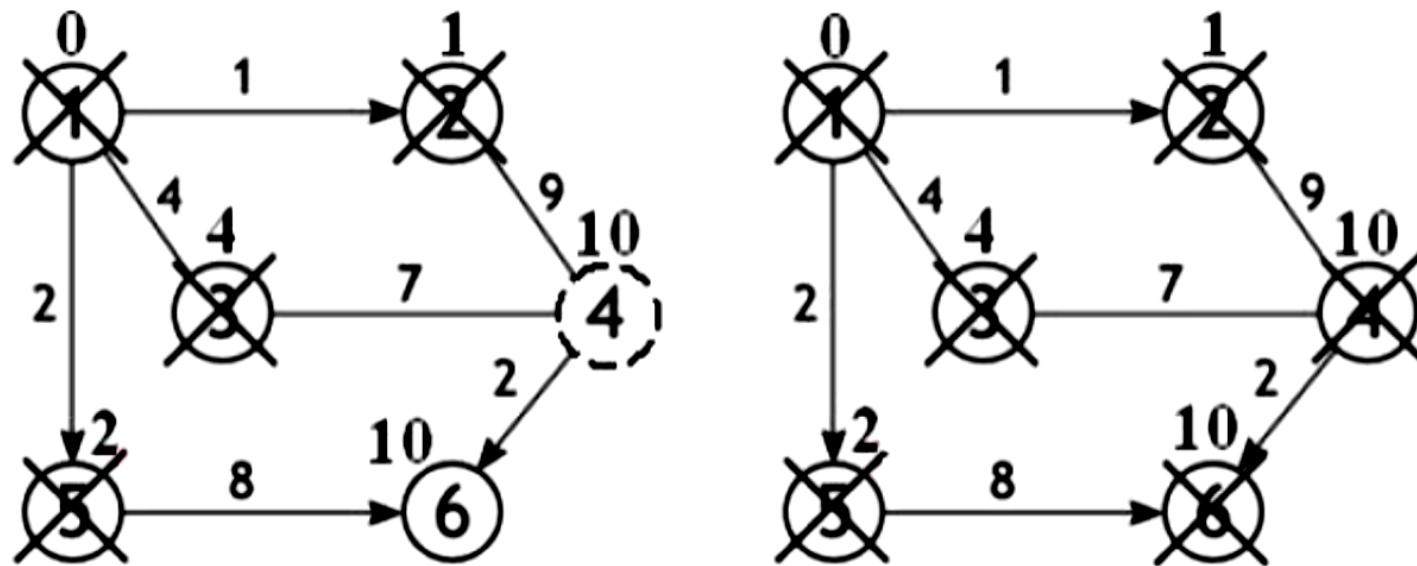


Рисунок 12.17

Когда все вершины графа, либо все те, что доступны из истока будут помечены как посещенные, тогда работа алгоритма Дейкстры завершится и все найденные пути будут кратчайшими. Так, например, будет выглядеть список самых оптимальных расстояний лежащих между вершиной 1 и всеми остальными вершинами, рассматриваемого графа:

$$1 \rightarrow 1 = 0$$

$$1 \rightarrow 2 = 1$$

$$1 \rightarrow 3 = 4$$

$$1 \rightarrow 4 = 10$$

$$1 \rightarrow 5 = 2$$

$$1 \rightarrow 6 = 10$$

В программе, находящий ближайшие пути между вершинами посредством метода Дейкстры, граф будет представлен в виде не бинарной матрицы смежности. Вместо единиц в ней будут выставлены веса ребер, функция нулей останется прежней: показывать, между какими вершинами нет ребер или же они есть, но отрицательно направленны.

Код программы на C++:

```
#include <iostream>
using namespace std;
const int V=6;
//алгоритм Дейкстры
void Dijkstra(int GR[V][V], int st){
    int distance[V], count, index, i, u, m=st+1;
    bool visited[V];
    for (i=0; i<V; i++) {
        distance[i]=INT_MAX; visited[i]=false;
    }
    distance[st]=0;
    for (count=0; count<V-1; count++) {
        int min=INT_MAX;
        for (i=0; i<V; i++)
            if (!visited[i] && distance[i]<=min){
                min=distance[i]; index=i;
            }
    }
}
```

```
        u=index;
        visited[u]=true;
        for (i=0; i<V; i++)
            if (!visited[i] && GR[u][i] && dis-
tance[u]!=INT_MAX &&
                distance[u]+GR[u][i]<distance[i])
                distance[i]=distance[u]+GR[u][i];
    }
    cout<<"Стоимость пути из начальной вершины до
остальных:\t\n";
    for (i=0; i<V; i++) if (distance[i]!=INT_MAX)
        cout<<m<<" > "<<i+1<<" = "<<distance[i]<<endl;
    else cout<<m<<" > "<<i+1<<" = "<<"маршрут
недоступен"<<endl;
}
```

```
//главная функция
void main(){
    setlocale(LC_ALL, "Rus");
    int start, GR[V][V]={
        {0, 1, 4, 0, 2, 0},
        {0, 0, 0, 9, 0, 0},
        {4, 0, 0, 7, 0, 0},
        {0, 9, 7, 0, 0, 2},
        {0, 0, 0, 0, 0, 8},
        {0, 0, 0, 0, 0, 0}};
    cout<<"Начальная вершина >> "; cin>>start;
    Dijkstra(GR, start-1);
    system("pause");
}
```