Министерство науки и высшего образования Российской Федерации

Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
**«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»**
(КФ МГТУ им. Н.Э. Баумана)

Ю.С. Белов, С.С. Гришунов

РЕАЛИЗАЦИЯ ОСНОВНЫХ АЛГОРИТМОВ С ГРАФАМИ

Методические указания к выполнению лабораторной работы
по курсу «Типы и структуры данных»

Калуга – 2019

УДК 004.62
ББК 32.972.5
Б435

Методические указания составлены в соответствии с учебным планом КФ МГТУ им. Н.Э. Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий».

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий» (ИУ4-КФ) протокол № 51.4/5 от «23» января 2019 г.

Зав. кафедрой ИУ4-КФ _____ к.т.н., доцент Ю.Е. Гагарин

- Методической комиссией факультета ИУ-КФ протокол № 7 от «28» 01 2019 г.

Председатель методической
комиссии факультета ИУ-КФ _____ к.т.н., доцент М.Ю. Адкин

- Методической комиссией
КФ МГТУ им.Н.Э. Баумана протокол № 4 от «5» 02 2019 г.

Председатель методической комиссии
КФ МГТУ им.Н.Э. Баумана _____ д.э.н., профессор О.Л. Перерва

Рецензент:
к.т.н., доцент кафедры ИУ6-КФ _____ А.Б. Лачихина

Авторы
к.ф.-м.н., доцент кафедры ИУ4-КФ _____ Ю.С. Белов
ассистент кафедры ИУ4-КФ _____ С.С. Гришунов

Аннотация

Методические указания к выполнению лабораторной работы по курсу «Типы и структуры данных» содержат общие сведения о графах, представлении графов в памяти компьютера и реализации основных алгоритмов над графами.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Типы и структуры данных» на кафедре «Программное обеспечение ЭВМ, информационные технологии» факультета «Информатика и управление» Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания содержат краткую теоретическую часть, описывающую работу основных алгоритмов над графами.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

Целью выполнения лабораторной работы является формирование практических навыков создания алгоритмов обработки графов.

Основными задачами выполнения лабораторной работы являются:

1. Познакомиться со способами представления графов в памяти компьютера.
2. Изучить основные обходы графов.
3. Научиться составлять алгоритмы для нахождения кратчайших путей в графе.
4. Реализовать алгоритм согласно варианту.

Результатами работы являются:

- Программа, реализующая индивидуальное задание
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Основные сведения о графах

Граф – конечное множество вершин и ребер, соединяющих их, т.е.: $G = \langle V, E \rangle$, где V – конечное непустое множество вершин; E – множество ребер (пар вершин). Если пары E (ребра) имеют направление, то граф называется *ориентированным* (орграф) (см. рис. 1), если иначе - *неориентированный* (неорграф). Если в графе встречаются однонаправленный и двунаправленные ребра, то граф называют *смешанным* (см. рис. 2). Если в пары E входят только различные вершины, то в графе нет *петель*. Если ребро графа имеет вес, то граф называется *взвешенным* (см. рис. 3). *Степень* вершины графа равна числу ребер, входящих и выходящих из нее (инцидентных ей). Если ребра инцедентны одной и той же паре вершин, то такие ребра называют *кратными*. Граф с кратными ребрами называют *мультиграфом* (см. рис. 4). Неорграф называется *связным*, если существует путь из каждой вершины в любую другую.

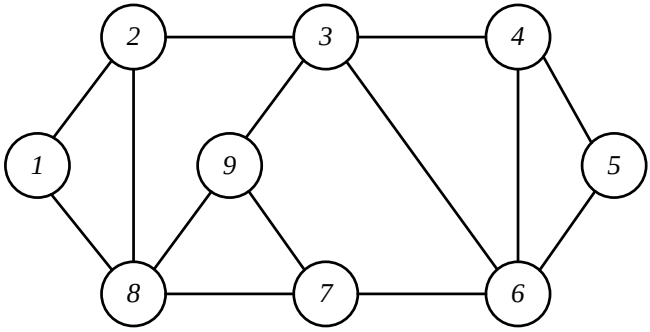


Рис. 1. Неориентированный граф

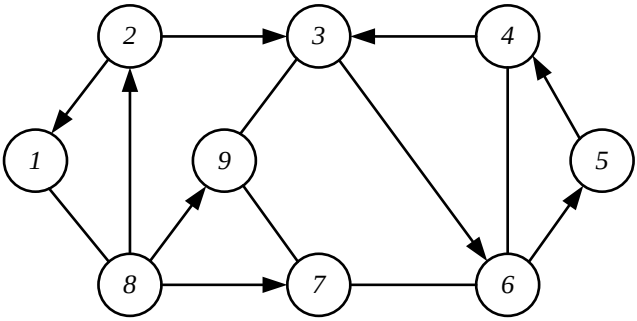


Рис. 2. Смешанный граф

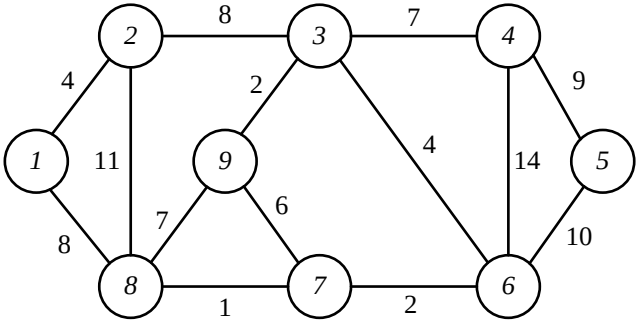


Рис. 3. Взвешенный граф

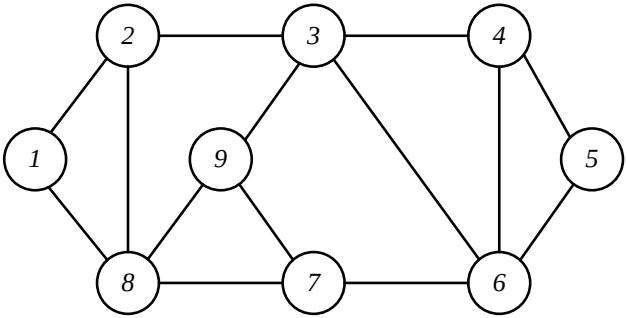


Рис. 4. Мультиграф

Обозначим количество вершин как $n = |V|$, а количество ребер как $m = |E|$. n называют *порядком* графа, m – *размером* графа.

Гамильтонов граф – граф, в котором существует цикл, содержащий все вершины графа по одному разу.

Гамильтонов путь – простой путь в графе, содержащий все вершины графа ровно по одному разу.

Дерево – связный граф без циклов.

Маршрут в графе – путь, ориентацией дуг которого можно пренебречь.

Простой путь – путь, все ребра которого попарно различны. Другими словами, простой путь не проходит дважды через одно ребро.

Маршрут, в котором все вершины попарно различны, называют *простой цепью*. Цикл, в котором все вершины, кроме первой и последней, попарно различны, называются простым циклом.

Путь – последовательность ребер (в неориентированном графе) и/или дуг (в ориентированном графе), такая, что конец одной дуги (ребра) является началом другой дуги (ребра). Или последовательность вершин и дуг (ребер), в которой каждый элемент инцидентен предыдущему и последующему. Может рассматриваться как частный случай маршрута.

Цепь в графе – маршрут, все ребра которого попарно различны.

Цикл – замкнутая цепь. Для орграфов цикл называется контуром.

Эйлеров граф – это граф, в котором существует цикл, содержащий все ребра графа по одному разу (вершины могут повторяться). Для существования эйлерова пути в связном графе необходимо и достаточно, чтобы граф содержал не более двух вершин нечетной степени.

Эйлерова цепь (или *эйлеров цикл*) – это цепь (цикл), которая содержит все ребра графа (вершины могут повторяться).

Представление графов в памяти компьютера

Графы в памяти могут представляться различными способами. Один из способов представления графов – это *матрица смежности* $A=(n\times n)$. В этой матрице элемент $a[i,j]=1$, если ребро, связывающее вершины V_i и V_j существует и $a[i,j]=0$, если ребра нет.

У неориентированных графов матрица смежности всегда симметрична. Построим матрицу смежности для графа с рисунка 1.

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{vmatrix}$$

В случае взвешенных графов строят весовую матрицу по такому же принципу, как и матрицу смежности, только вместо 1 в качестве значения $a[i, j]$ записывают вес ребра между вершинами i и j. Рассмотрим весовую матрицу графа для рисунка 3.

$$A = \begin{vmatrix} 0 & 4 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 4 & 0 & 8 & 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 8 & 0 & 7 & 0 & 4 & 0 & 0 & 2 \\ 0 & 0 & 7 & 0 & 9 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 4 & 14 & 10 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 1 & 6 \\ 8 & 11 & 0 & 0 & 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 2 & 0 & 0 & 0 & 6 & 7 & 0 \end{vmatrix}$$

Во многих случаях удобнее представлять граф в виде так называемого *списка смежностей*. Список смежностей содержит для каждой вершины из множества вершин V список тех вершин, которые непосредственно связаны с этой вершиной. Каждый элемент списка смежностей является записью, содержащей данную вершину и указатель на следующую запись в списке (для последней записи в списке этот указатель – пустой). Входы в списки смежностей для каждой вершины графа хранятся в отдельной таблице (массиве).

Например, для графа с рисунка 1 список смежностей выглядит следующим образом (рис. 5):

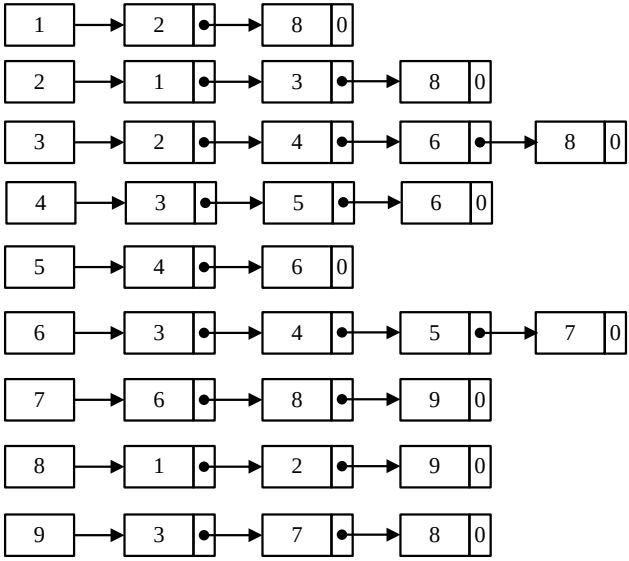


Рис. 5. Список смежностей

Еще одним способом представления графа в памяти компьютера является построение *матрицы инцендентности*. В данной матрице указываются связи между инцендентными вершинами и ребрами. Строки в данной матрице соответствуют вершинам, столбцы – ребрам. Элемент $a[i, j] = 1$, если вершина i инцендента ребру j.

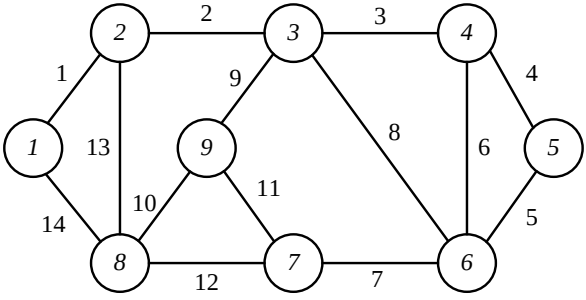


Рис. 6. Неориентированный граф

Например, для графа с рисунка 6 матрица инцендентности будет

выглядеть следующим образом.

$$A = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{vmatrix}$$

Рассмотрим реализацию графа, представленного как отдельная сущность в виде класса, на языке C++.

```
class Graph
{
   public:
   int A[7][7];  //матрица смежности
   int str;      //размер матрицы

   Graph()       //стандартный конструктор графа
   {
       this->str = 7;
       for(int i = 0; i < this->str; i++)
           for(int j = i; j < this->str; j++)
           {
               this->A[i][j] = 0;
               this->A[j][i] = this->A[i][j];
           }
   }
   Graph(int _n) //конструктор графа
   {
       this->str = _n;
       for(int i = 0; i < this->str; i++)
           for(int j = i; j < this->str; j++)
           {
               this->A[i][j] = 0;
               this->A[j][i] = this->A[i][j];
           }
   }

   //обращение к элементу матрицы
```

```
    int & operator () (int _i, int _j)
    {
        return this->A[_i][_j];
    }

    void setMatrix(int _n)    //изменение матрицы
    {
        this->str = _n;
    }

    void ShowGraph()    //показ матрицы
    {
        system("cls");
        cout<<"Матрица смежности:\n";
        for(int i=0; i<str; i++)
        {
            for( int j=0; j<str; j++)
            {
            cout<<" "<<A[i][j]<<" ";
            }
            cout<<"\n";
        }
    }
}
```

Обходы графов

В основе построения большинства алгоритмов на графах лежит систематический перебор вершин графа, при котором каждая вершина просматривается в точности один раз, а количество просмотров ребер графа ограничено заданной константой (лучше – не более одного раза). Данная операция называется обходом графа. Рассмотрим два классических алгоритма обхода графа.

Алгоритм поиска в глубину

Один из основных методов проектирования графовых алгоритмов – это *поиск* (или обход графа) *в глубину* (depth first search, DFS), при котором, начиная с произвольной вершины v_0, ищется ближайшая смежная вершина *v*, для которой в свою очередь осуществляется поиск в глубину (т.е. снова ищется ближайшая, смежная с ней вершина) до тех пор, пока не встретится ранее просмотренная вершина, или не

закончится список смежности вершины v (то есть вершина полностью обработана). Если нет новых вершин, смежных с v, то вершина v считается использованной, идет возврат в вершину, из которой попали в вершину v, и процесс продолжается до тех пор, пока не получим $v = v_0$. Иными словами, поиск в глубину из вершины v основан на поиске в глубину из всех новых вершин, смежных с вершиной v.

Путь, полученный методом поиска в глубину, в общем случае не является кратчайшим путем из одной вершины в другую. Это является его недостатком.

Рассмотрим выполнение алгоритма на примере (см. рис. 7).

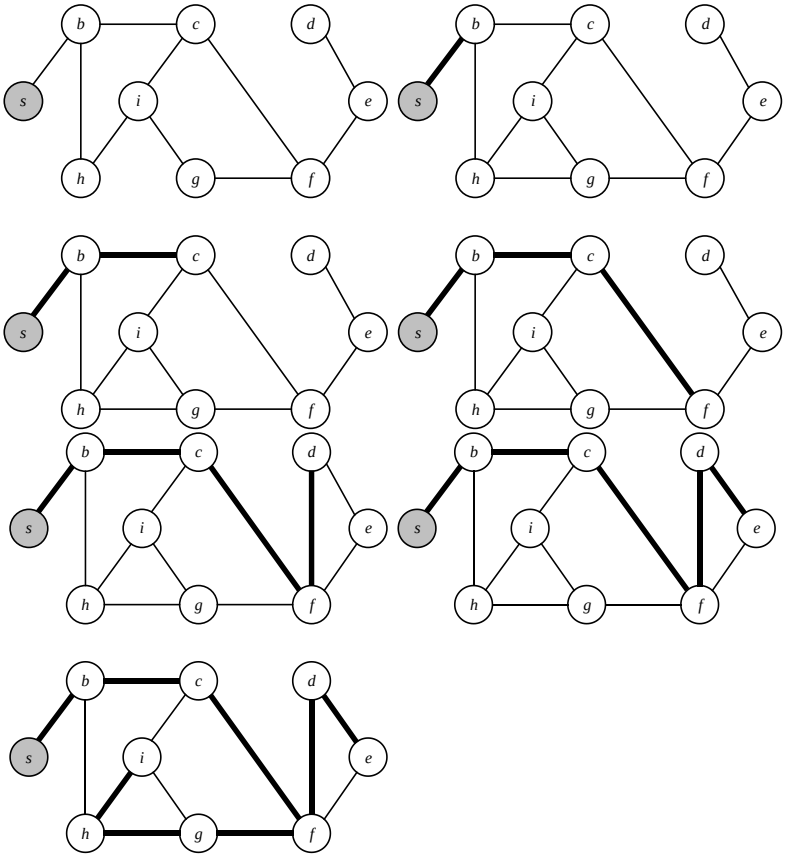


Рис. 7. Выполнение алгоритма DFS

Реализация алгоритма описка в глубину на языке C++:

```
   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);
}
```

Алгоритм поиска в ширину

Указанного недостатка лишен другой метод обхода графа – *поиск в ширину* (breadth first search, BFS). Обработка вершины *v* осуществляется путем просмотра сразу всех новых соседей этой вершины. При этом полученный путь является кратчайшим путем из одной вершины в другую.

Рассмотрим выполнение алгоритма на примере (см. рис. 8).

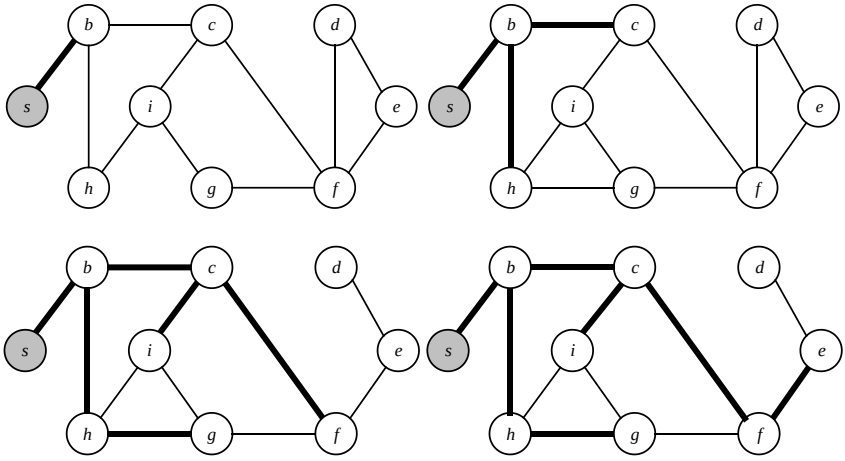


Рис. 8. Выполнение алгоритма BFS

Реализация алгоритма описка в ширину на языке C++:

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

Нахождение кратчайшего пути

Модели взвешенных графов, в которых с каждым ребром ассоциирован его вес (*weight*) или стоимость (*cost*), используются во многих приложениях. В картах авиалиний, в которых ребрами отмечены авиарейсы, а их веса означают расстояния или стоимости билетов. В задачах календарного планирования веса могут представлять время или трудоемкость выполнения задачи. В таких ситуациях естественно возникают вопросы минимизации затрат.

Поиск кратчайших путей до всех вершин из одной указанной вершины для взвешенного орграфа с неотрицательными ребрами осуществляется с использованием алгоритма *Дейкстры*.

Алгоритм *Беллмана-Форда* позволяет решить задачу о поиске кратчайших путях из одной выбранной вершины ко всем остальным вершинам при любых весах ребер, в том числе и отрицательных.

Алгоритм Дейкстры

Данный алгоритм является алгоритмом на графах, который

изобретен нидерландским ученым Э. Дейкстрой в 1959 году. Алгоритм находит кратчайшее расстояние от одной из вершин графа до всех остальных и работает только для графов без ребер отрицательного веса.

Каждой вершине приписывается вес – это вес пути от начальной вершины до данной. Также каждая вершина может быть выделена. Если вершина выделена, то путь от нее до начальной вершины кратчайший, если нет – то временный. Обходя граф, алгоритм считает для каждой вершины маршрут, и, если он оказывается кратчайшим, выделяет вершину. Весом данной вершины становится вес пути. Для всех соседей данной вершины алгоритм также рассчитывает вес, при этом ни при каких условиях не выделяя их. Алгоритм заканчивает свою работу, дойдя до конечной вершины, и весом кратчайшего пути становится вес конечной вершины.

Описание алгоритма

Шаг №1. Всем вершинам, за исключением первой, присваивается вес равный бесконечности, а первой вершине – 0.

Шаг №2. Все вершины не выделены.

Шаг №3. Первая вершина объявляется текущей.

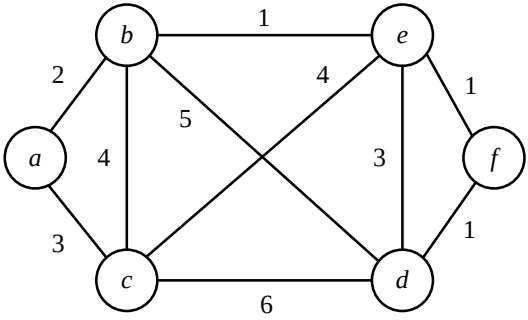
Шаг №4. Вес всех невыделенных вершин пересчитывается по формуле: вес невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса ребра, соединяющего текущую вершину с невыделенной.

Шаг №5. Среди невыделенных вершин ищется вершина с минимальным весом. Если таковая не найдена, то есть вес всех вершин равен бесконечности, то маршрут не существует. Следовательно, выход. Иначе, текущей становится найденная вершина. Она же выделяется.

Шаг №6. Если текущей вершиной оказывается конечная, то путь найден, и его вес есть вес конечной вершины.

Шаг №7. Переход на шаг 4.

Рассмотрим пример для следующего графа. Построим матрицу согласно алгоритму, определяющую кратчайший путь от вершины *a* до всех остальных вершин.



Вершины	a	b	c	d	e	f
a	0	2a	3a	∞	∞	∞
b		***2a***	3a	7b	3b	∞
c			***3a***	7b	3b	∞
e				6e	***3b***	4e
f				5f		***4e***
d				***5f***		

Рис. 9. Пример применения алгоритма Дейкстры

Алгоритм Дейкстры на языке C++:

```
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]   &&
            distance[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;
}
```

Алгоритм Беллмана-Форда

Алгоритм решает ту же задачу, что алгоритм Дейкстры, но работает в графах с отрицательными дугами и позволяет обнаруживать отрицательные циклы. Алгоритм проще в реализации, но хуже в производительности.

Решить задачу, т. е. найти все кратчайшие пути из вершины s до всех остальных, используя алгоритм Беллмана-Форда, это значит воспользоваться методом динамического программирования: разбить ее на типовые подзадачи, найти решение последним, покончив тем самым с основной задачей. Здесь решением каждой из таких подзадач является определение наилучшего пути от одного отдельно взятого ребра, до какого-либо другого. Для хранения результатов работы алгоритма заведем одномерный массив $d[]$. В каждом его i-ом элементе будет храниться значение кратчайшего пути из вершины s до вершины i (если таковое имеется). Изначально, присвоим элементам массива $d[]$ значения равные условной бесконечности (например,

число заведомо большее суммы всех весов), а в элемент $d[s]$ запишем нуль. Так мы задействовали известную и необходимую информацию, а именно известно, что наилучший путь из вершины s в нее же саму равен 0, и необходимо предположить недоступность других вершин из s. По мере выполнения алгоритма, для некоторых из них, это условие окажется ложным, и вычисляться оптимальные стоимости путей до этих вершин из s.

Задан граф $G = \langle V, E \rangle$, $n = |V|$, а $m = |E|$. Обозначим смежные вершины этого графа символами v и u, а вес ребра (v,u) символом w. Иначе говоря, вес ребра, выходящего из вершины v и входящего в вершину u, будет равен w. Тогда ключевая часть алгоритма Беллмана-Форда примет следующий вид:

```
Для i от 1 до n-1 выполнять
    Для j от 1 до m выполнять
          Если d[v] + w(v, u) < d[u] то
             d[u] = d[v] + w(v, u)
```

На каждом n-ом шаге осуществляются попытки улучшить значения элементов массива $d[]$: если сумма, составленная из веса ребра $w(v,u)$ и веса хранящегося в элементе $d[v]$, меньше веса $d[u]$, то она присваивается последнему.

Алгоритм Беллмана-Форда и его демонстрация в основной программе на языке C++:

```
struct Edge
{
   int a, b, w;
};

const int inf = 1000;
Edge edges[10];
int d[10];
int i, j, n, start, e, w;
```

```
void BellmanFordAlgorithm(int _n, int _start)
{
	for (i = 0; i < n; i++)
		d[i] = inf;
	d[start] = 0;

	cout << "Before:\nd:\t";
	for (int k = 0; k < n; k++)
	   cout << d[k] << "\t";
	cout << endl;
	for (i = 0; i < n; i++)
	{
	   if (i < n-1)
	   {
	      for (j = 0; j < e; j++)
	      {
	         if(d[edges[j].a]+edges[j].w < d[edges[j].b])
	         {
	            d[edges[j].b]= d[edges[j].a]+ edges[j].w;
	         }
	      }
	   }
	   cout << "\nAfter iteration:\td:\t";
	   for (int k = 0; k < n; k++)
	      cout << d[k] << "\t";
	   if (i == n-1)
	   {
	      for (j = 0; j < e; j++)
	      {
	         if(d[edges[j].a] +edges[j].w <d[edges[j].b])
	         {
	             cout << "\nThere is a negative cycle!";
	             break;
	         }
	      }
	   }
	}
	system("pause");
	for (i = 0; i < n; i++)
	   if (d[i] == inf)
	      cout<< endl<< _start<< "->" <<i <<" = NOT";
	   else
	      cout<< endl<< _start<< "->" <<i <<" = " <<d[i];
	system("pause");
};
```

```
void main()
{
   cout << "Enter vertex count: ";
   cin >> n;
   e = 0;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
      {
         cout<<"Enter weight between "<<i<<" and "<<j<<": ";
         cin >> w;
         if (w)
         {
            edges[e].a = i;
            edges[e].b = j;
            edges[e].w = w;
            e++;
         }
      }
   for (i = 0; i < e; i++)
      cout<<edges[i].a<<"|"<<edges[i].b<<"|"<<edges[i].w<<"\t";

   cout << "\nEnter start:";
   cin >> start;
   BellmanFordAlgorithm(n, start);
   cout << "\n";
   system("pause");
}
```

Алгоритм Флойда-Уоршелла

Дан ориентированный или неориентированный взвешенный граф G с n вершинами (см. рис. 10). Требуется найти значения всех величин w_{ij} – длины кратчайшего пути из вершины i в вершину j.

Предполагается, что граф не содержит циклов отрицательного веса.

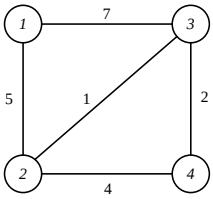


Рис. 10. Пример графа

Ключевая идея алгоритма – разбиение процесса поиска кратчайших путей на итерации.

Перед k-ой итерацией $(k = 1..n)$ считается, что в матрице расстояний $w[][]$ сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $[1,..,k-1]$ (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой итерацией величина $w[i][j]$ равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой итерации, достаточно в матрицу расстояний $w[][]$ записать матрицу смежности графа: $w[i][j]$ – стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" (∞). Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Для восстановления самого пути между любыми двумя заданными вершинами будем использовать еще одну матрицу историй $h[][]$, которая для каждой пары вершин будет содержать номер фазы, на которой было получено кратчайшее расстояние между ними. Изначально мы ее заполняем по следующему закону: $h[i][j] = j$

Пусть теперь мы находимся на k-ой итерации, и хотим пересчитать матрицу $w[][]$ таким образом, чтобы она соответствовала требованиям уже для $k+1$-ой фазы.

Будем пользоваться следующими соотношениями:

Если $w_{ij}^{k-1} > w_{ik}^{k-1} + w_{kj}^{k-1}$, то $w_{ij}^{k} = w_{ik}^{k-1} + w_{kj}^{k-1}$ и $h_{ij}^{k} = k$, иначе $w_{ij}^{k} = w_{ij}^{k-1}$ и $h_{ij}^{k} = h_{ij}^{k-1}$.

Для графа с рисунка 10 процесс получения кратчайших расстояний будет выглядеть следующим образом.

W_0

в.	1	2	3	4
1	0	5	7	∞
2	5	0	1	4
3	7	1	0	2
4	∞	4	2	0

H_0

в.	1	2	3	4
1	0	2	3	4
2	1	0	3	4
3	1	2	0	4
4	1	2	3	0

I итерация

W_1

в.	1	2	3	4
1	0	5	7	∞
2	5	0	1	4
3	7	1	0	2
4	∞	4	2	0

H_1

в.	1	2	3	4
1	0	2	3	4
2	1	0	3	4
3	1	2	0	4
4	1	2	3	0

II итерация

W_2

в.	1	2	3	4
1	0	5	***6***	***9***
2	5	0	1	4
3	***6***	1	0	2
4	***9***	4	2	0

H_2

в.	1	2	3	4
1	0	2	***2***	***2***
2	1	0	3	4
3	***2***	2	0	4
4	***2***	2	3	0

III итерация

W_3

в.	1	2	3	4
1	0	5	6	***8***
2	5	0	1	***3***
3	6	1	0	2
4	***8***	***3***	2	0

H_3

в.	1	2	3	4
1	0	2	2	***3***
2	1	0	3	***3***
3	2	2	0	4
4	***3***	***3***	3	0

IV итерация

W_4

в.	1	2	3	4
1	0	5	6	8
2	5	0	1	3
3	6	1	0	2
4	8	3	2	0

H_4

в.	1	2	3	4
1	0	2	2	3
2	1	0	3	3
3	2	2	0	4
4	3	3	3	0

Например, мы хотим узнать кратчайший путь из вершины 1 в 4. Из матрицы весов мы видим, что расстояние равно 8. А из матрицы истории мы можем увидеть, какие вершины нам помогут добраться из вершины 1 до вершины 4 за расстояние 8. Рассмотрим 1-ю строку: 1→4

= 3, 1→3 = 2, 1→2 = 2. Следовательно, путь будет следующим: 1→2→3→4.

Реализация алгоритма Флойда-Уоршелла на языке C++:

```
void FU(int D[][maxV], int V)
{
    int k;
    for (i=0; i<V; i++) D[i][i]=0;

    for (k=0; k<V; k++)
          for (i=0; i<V; i++)
                for (j=0; j<V; j++)
                      if (D[i][k] && D[k][j] && i!=j)
                         if (D[i][k]+D[k][j]<D[i][j]
                            || D[i][j]==0)
                           D[i][j]=D[i][k]+D[k][j];
    for (i=0; i<V; i++)
    {
          for (j=0; j<V; j++) cout<<D[i][j]<<"\t";
          cout<<endl;
    }
}
```

Остовные деревья графов

Остовное дерево – ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины. Неформально говоря, остовное дерево состоит из некоторого подмножества ребер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим ребрам, и в нем нет циклов, то есть из любой вершины нельзя попасть в саму себя, не пройдя какое-то ребро дважды.

Минимальное остовное дерево (*MST* – minimal spanning tree, минимальный остов, минимальный каркас) взвешенного графа – это остовное дерево, сумма весов его ребер которого не превосходит вес любого другого остовного дерева этого графа (рис. 11).

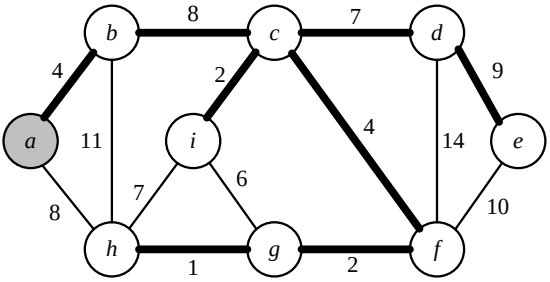


Рис. 11. MST-дерево графа

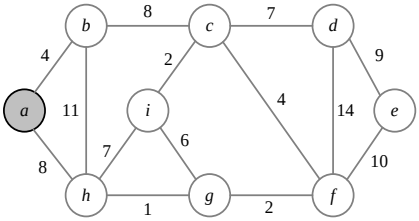
Задача поиска минимального остовного дерева в произвольном взвешенном неориентированном графе применяется во многих ситуациях. Наиболее часто для построения остовых деревьев минимальной стоимости используют два классических алгоритма: алгоритм Крускала и алгоритм Прима. Рассмотрим их подробнее.

Алгоритм Прима

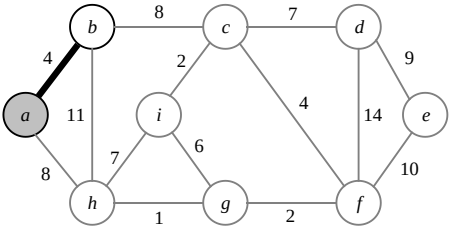
Алгоритм Прима обладает тем свойством, что ребра во множестве E всегда образуют единое дерево. Дерево начинается с произвольной корневой вершины a и растет до тех пор, пока не охватит все вершины в V. На каждом шаге к дереву добавляется легкое ребро, соединяющее дерево и отдельную вершину из оставшейся части графа. Данное правило добавляет только безопасные для дерева ребра; следовательно, по завершении алгоритма ребра образуют минимальное остовное дерево. Данная стратегия является жадной, поскольку на каждом шаге к дереву добавляется ребро, которое вносит минимально возможный вклад в общий вес.

Выполнение алгоритма Прима:

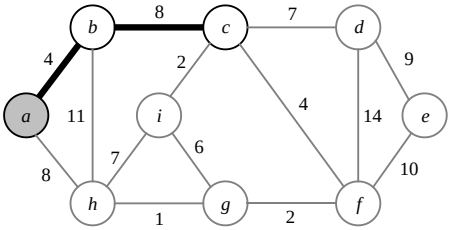
В качестве входных данных алгоритму передаются связный граф G и корень a минимального остовного дерева.



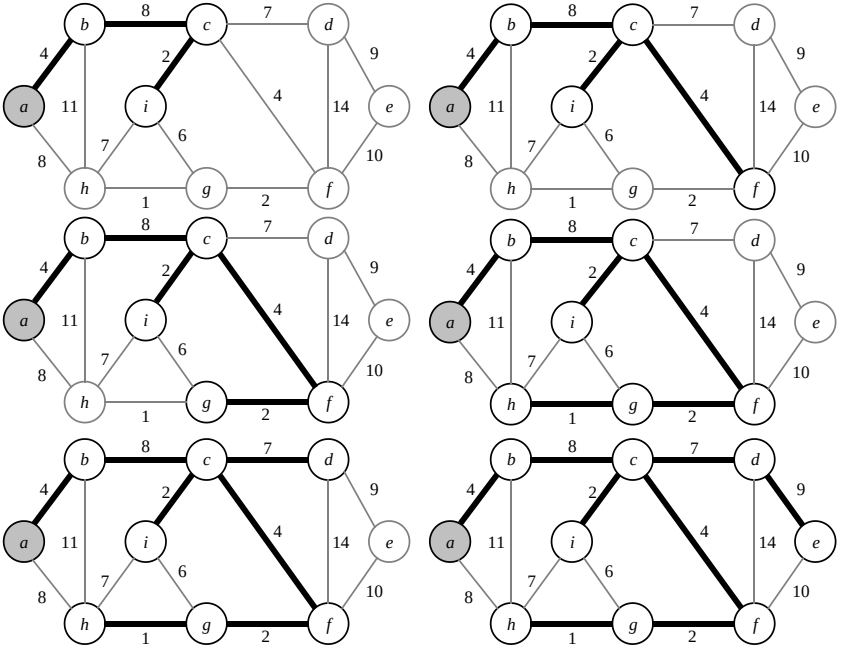
1. Ребро с весом 4 является минимальным и добавляется к дереву.



2. Далее из ребер, смежных с a и b, с весами 8, 11, 8 выбираем безопасное и добавляем к дереву.



3. Добавляем остальные безопасные ребра.



Рассмотрим реализацию алгоритма Прима на языке C++:

```
// входные данные
int n;
vector < vector<int> > g;
const int INF = 1000000000;//значение "бесконечность"

// алгоритм
vector<bool> used (n);
vector<int> min_e (n, INF), sel_e (n, -1);
min_e[0] = 0;
for (int i=0; i<n; ++i)
{
    int v = -1;
    for (int j=0; j<n; ++j)
        if (!used[j] && (v== -1|| min_e[j]<min_e[v]))
            v = j;
    if (min_e[v] == INF)
    {
        cout << "No MST!";
        exit(0);
    }

    used[v] = true;
    if (min_e[v] != -1)
        cout << v << " " << min_e[v] << endl;

    for (int to=0; to<n; ++to)
        if (g[v][to] < min_e[to])
        {
            min_e[to] = g[v][to];
            sel_e[to] = v;
        }
}
```

Алгоритм Крускала

Алгоритм Крускала строит MST-дерево по одному ребру, находя на каждом шаге наименьшее ребро, которое присоединяется к единственно растущему дереву. В отличие от алгоритма Прима, он отыскивает ребро, которое соединяет два дерева в лесу. Построение начинается с вырожденного леса из V деревьев (каждое состоящее из одной вершины), а затем выполняется объединение двух деревьев

самым коротким ребром, пока не останется единственное дерево – MST.

Выполнение алгоритма Крускала:

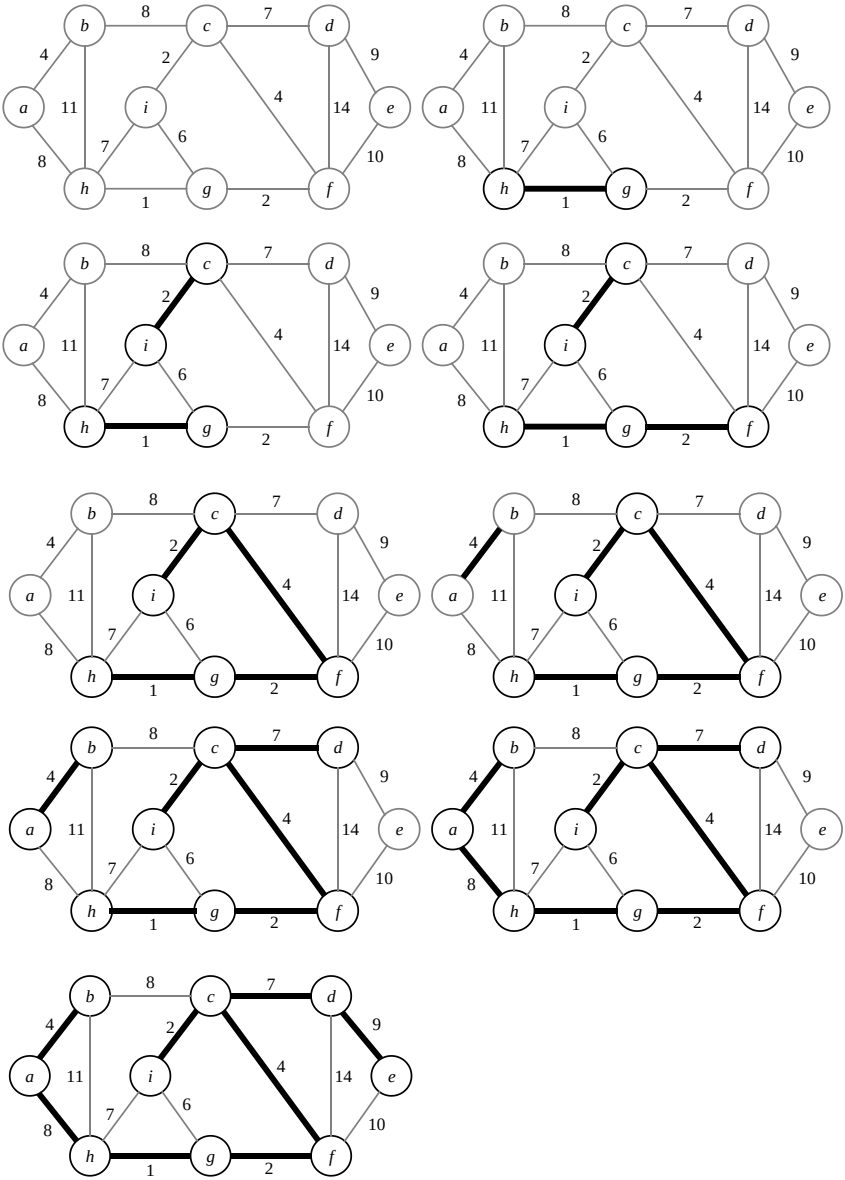


Рис 12. Выполнение алгоритма Крускала

Алгоритм Крускала на языке C++:

```
    int m;
    // вес - вершина 1 - вершина 2
    vector <pair<int, pair<int,int>>> g (m);
    int cost = 0;
    vector < pair<int,int> > res;

    sort (g.begin(), g.end());
    vector<int> tree_id (n);
    for (int i=0; i<n; ++i)
        tree_id[i] = i;
    for (int i=0; i<m; ++i)
    {
        int a=g[i].second.first, b=g[i].second.second,
            l=g[i].first;
        if (tree_id[a] != tree_id[b])
        {
              cost += l;
              res.push_back (make_pair (a, b));
              int old_id = tree_id[b], new_id=tree_id[a];
              for (int j=0; j<n; ++j)
                    if (tree_id[j] == old_id)
                          tree_id[j] = new_id;
        }
}
```

ЗАДАЧИ НА ГРАФАХ

Задача №1: Дан неориентированный граф G=(V,E). Необходимо проверить, является ли он связным.

Если пары E (ребра) не имеют направлений, то граф называется неориентированный (неорграф). Неорграф называется *связным*, если существует путь из каждой вершины в любую другую.

Для решения данной задачи следует вспомнить один из основных методов проектирования графовых алгоритмов – это *поиск* (или обход графа) *в глубину* (depth first search, DFS), при котором, начиная с произвольной вершины v_0, ищется ближайшая смежная вершина v, для которой в свою очередь осуществляется поиск в глубину (т.е. снова ищется ближайшая, смежная с ней вершина) до тех пор, пока не встретится ранее просмотренная вершина, или не закончится список смежности вершины v (то есть вершина полностью обработана). Если нет новых вершин, смежных с v, то вершина v считается использованной, идет возврат в вершину, из которой попали в вершину v, и процесс продолжается до тех пор, пока не получим $v = v_0$. Иными словами, поиск в глубину из вершины v основан на поиске в глубину из всех новых вершин, смежных с вершиной v.

Пошаговое представление алгоритма обхода в глубину

1. Выбираем любую вершину из еще *не пройденных*, обозначим ее как uu.
2. Запускаем процедуру dfs(u)dfs(u)
   - Помечаем вершину uu как *пройденную*
   - Для каждой *не пройденной* смежной с uu вершиной (назовем ее vv) запускаем dfs(v)dfs(v)
3. Повторяем шаги 1 и 2, пока все вершины не окажутся *пройденными*.

Прежде чем мы перейдем к реализации определим как мы будем хранить граф в памяти компьютера. В данном случае удобнее всего хранить в виде матрицы смежности:

```
#define GRAPH_SIZE 5
typedef int graph[GRAPH_SIZE][GRAPH_SIZE];
graph G;
```

Теперь следует определить алгоритмы ввода/вывода на графах, ведь считывать граф мы будем из файла:

```
	void ReadGraphFromFile(graph &G)
	{
	 int x, y;
	 std::ifstream in("graph.txt");

	 if (!in) {
		std::cout << "Cannot open file.\n";
		return;
	 }

	 for (y = 0; y < GRAPH_SIZE; y++) {
		for (x = 0; x < GRAPH_SIZE; x++) {
			in >> G[x][y];
		}
	 }

	 in.close();
	}

	void PrintGraph(graph G)
	{
	 int x, y;
	 for (y = 0; y < GRAPH_SIZE; y++) {
		for (x = 0; x < GRAPH_SIZE; x++) {
			std::cout << G[x][y] << " ";
		}
		std::cout << std::endl;
	 }
	 std::cout << std::endl;
}
```

Пусть матрица смежности графа задана в файле graph.txt:

```
0 0 1 0 0
0 0 1 0 0
1 1 0 1 0
0 0 1 0 1
0 0 0 1 0
```

Ее вид на диаграмме (рис. 13):

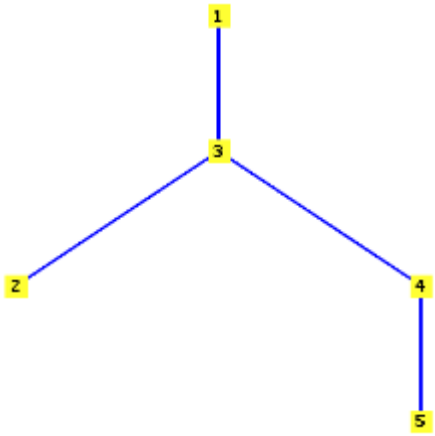


Рис. 13. Диаграмма исходного графа к задаче №1

Осталось реализовать алгоритм:

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

Однако классическая реализация нам не совсем подходит. Нам необходимо сделать небольшую модификацию алгоритма обхода в глубину, в которой уже будем возвращать количество посещенных вершин. Запустим такой dfs() от некоторой вершины графа GG, если его результат равен |V||V|, то мы побывали во всех вершинах графа, а следовательно он связен, иначе какие-то вершины остались непосещенными. Работает алгоритм за O(|V|+|E|).

```
int DFS(int st, bool* visited)
{
 int r;
 int visitedVertices = 1;
 std::cout << st + 1 << " ";
 visited[st] = true;
 for (r = 0; r <= GRAPH_SIZE; r++)
       if ((G[st][r] != 0) && (!visited[r]))
             visitedVertices += DFS(r, visited);
 return visitedVertices;
}
```

Теперь реализуем код самой программы

```
int main()
{
       ReadGraphFromFile(G);

       bool visited[GRAPH_SIZE];

       for (int i = 0; i < GRAPH_SIZE; i++)
       {
             visited[i] = false;
       }

       int result = DFS(0, visited);
       std::cout << std::endl;

       if (result == GRAPH_SIZE)
          std::cout << "Graph is connected" << std::endl;
       else
          std::cout << "Graph isn't connected"<< std::endl;

       system("pause");
        return 0;
}
```

Результат работы программы:

```
Graph is connected
```

Задача №2: Выдать минимальный каркас графа в виде списка входящих в него ребер, используя алгоритм Крускала.

Итак, дадим определения. *Каркас графа* – ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины. Неформально говоря, остовное дерево состоит из некоторого подмножества ребер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим ребрам, и в нем нет циклов, то есть из любой вершины нельзя попасть в саму себя, не пройдя какое-то ребро дважды. Минимальный каркас взвешенного графа – это остовное дерево, сумма весов его ребер которого не превосходит вес любого другого остовного дерева этого графа (рис. 14).

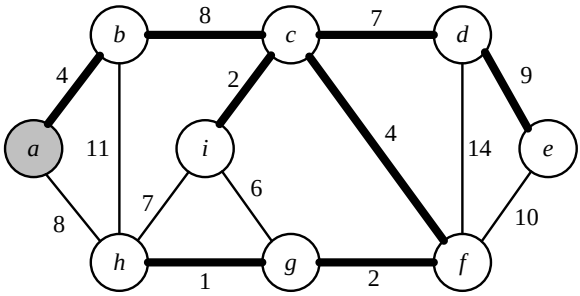


Рис. 14 Минимальный каркас графа

Для решения данной задачи нам необходимо вспомнить алгоритм Крускала. Алгоритм Крускала строит минимальный каркас графа по одному ребру, находя на каждом шаге наименьшее ребро, которое присоединяется к единственно растущему дереву. В отличие от алгоритма Прима, он отыскивает ребро, которое соединяет два дерева в лесу. Построение начинается с вырожденного леса из V деревьев (каждое состоящее из одной вершины), а затем выполняется объединение двух деревьев самым коротким ребром, пока не останется единственное дерево – минимальный каркас.

Перейдем к написанию программы.

Пусть граф задан матрицей смежности, находящейся в файле graph.txt:

```
 0 16 15 18 14 18 18 14
16  0 17 14 13 19 18 14
15 17  0 12 12 10 20 17
18 14 12  0 12 14 20 10
14 13 12 12  0 18 18 11
18 19 10 14 18  0 13 18
18 18 20 20 18 13  0 14
14 14 17 10 11 18 14  0
```

На диаграмме он будет выглядеть так (рис. 15):

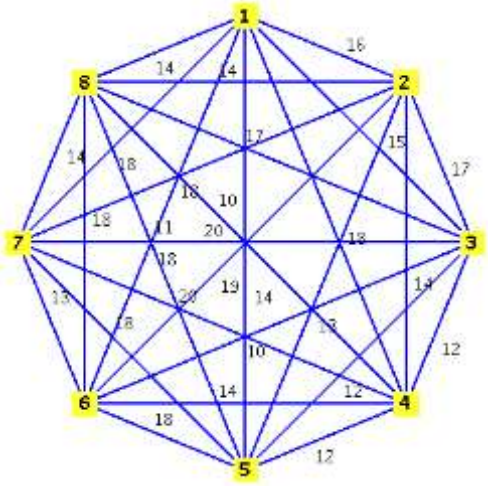


Рис. 15. Диаграмма исходного графа для задачи №2

Определим способ хранения матрицы в памяти компьютера

```
int m = 28; //количество пар вершин
int n = 8;  //количество вершин
std::vector <std::pair<int, std::pair<int, int>>> g(m);
// вес - вершина 1 - вершина 2

int cost = 0; //общий вес каркаса
std::vector <std::pair<int, int>> res; //результат
```

Далее следует реализовать чтение графа из файла

```
	void ReadGraphFromFile()
	{
	 int x, y;
	 std::ifstream in("graph.txt");

	 if (!in) {
		std::cout << "Cannot open file.\n";
		return;
	 }

	 int c = 0;
	 int temp = 0;
	 for (y = 0; y < n; y++) {
		for (x = 0; x < n; x++) {
			in >> temp;
			if (x > y)
			{
				g[c].first = temp;
				g[c].second.first = y;
				g[c].second.second = x;
				c++;
			}
		}
	 }
	 in.close();
}
```

Затем реализуем сам алгоритм

```
	ReadGraphFromFile();
	 std::sort(g.begin(), g.end());
	 std::vector<int> tree_id(n);
	 for (int i = 0; i<n; ++i)
		tree_id[i] = i;
	 for (int i = 0; i < m; i++)
	 {
		int a = g[i].second.first;
		int b = g[i].second.second;
		int l = g[i].first;
		if (tree_id[a] != tree_id[b])
		{
			cost += l;
			res.push_back(std::make_pair(a, b));
			int old_id = tree_id[b], new_id=tree_id[a];
```

```
                for (int j = 0; j<n; ++j)
                      if (tree_id[j] == old_id)
                            tree_id[j] = new_id;
          }
    }
```

Теперь все что осталось это вывести результаты работы

```
    std::cout << "Wieght: " << cost<<std::endl;
    std::cout << "Pairs:" << std::endl;
    for (int x = 0; x < 7; x++) {
          std::cout << res[x].first << "-";
          std::cout << res[x].second << std::endl;
    }
```

Результат выполнения программы:

```
Wieght: 83
Pairs:
2-5
3-7
4-7
2-3
1-4
5-6
0-4
```

Результат выполнения на диаграмме (рис. 16):

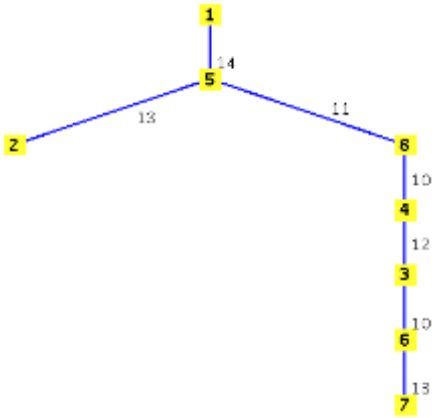


Рис. 16. Минимальный каркас исходного графа

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Создать программу согласно варианту, полученному у преподавателя. При выполнении лабораторной работы запрещается использовать сторонние классы и компоненты, реализующие заявленную функциональность.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту, использовать сторонние классы и компоненты, реализующие функционал задания запрещено.

ВАРИАНТЫ ЗАДАНИЙ

1. Найти все вершины заданного графа, недостижимые из заданной его вершины.
2. Определить, является ли связным заданный граф.
3. Для двух выделенных вершин графа построить соединяющий их простой путь.
4. Найти самый длинный простой путь в графе.
5. Найти все вершины графа, к которым от заданной вершины можно добраться по пути не длиннее A.
6. Задан граф – не дерево. Проверить, можно ли превратить его в дерево удалением одной вершины вместе с ее ребрами.
7. В графе найти максимальное расстояние между всеми парами его вершин.
8. Задана система односторонних дорог. Найти путь, соединяющий города A и B и не проходящий через заданное множество городов.
9. Задана система двусторонних дорог. Для каждой пары городов найти длину кратчайшего пути между ними.
10. Задана система двусторонних дорог. Найти два города и соединяющий их путь, который проходит через каждую из дорог системы только один раз.
11. Задана система двусторонних дорог. Найти замкнутый путь длиной не более T, проходящий через каждую дорогу ровно один раз.

12. Задана система двусторонних дорог, где для любой пары городов есть соединяющий их путь. Найти город с минимальной суммой расстояний до остальных городов.
13. Задана система двусторонних дорог. Определить, можно ли, построив еще три новые дороги, из заданного города добраться до каждого из остальных городов, проезжая расстояние не более T единиц.
14. Задана система двусторонних дорог. Определить, можно ли, закрыв какие-нибудь три дороги, добиться того, чтобы из города A нельзя было попасть в город B.
15. Задана система двусторонних дорог. Найти множество городов, расстояние от которых до выделенного города (столицы) больше, чем T.
16. Для заданного графа, используя метод поиска в глубину, определить длины всех путей из города A в город B.
17. Для заданного графа, используя метод поиска в ширину, определить длины всех путей из города A в город B.
18. Для заданного графа, используя метод поиска в глубину, определить кратчайшее расстояние из города A в город B.
19. Для заданного графа, используя метод поиска в ширину, определить кратчайшее расстояние из города N в город M.
20. Задана карта дорог в виде графа. Найти города, расположенные на максимальном и минимальном расстоянии от города A.
21. Задана карта дорог в виде графа. Найти города, расположенные на расстоянии не более T от города A.
22. Найти самый короткий простой путь в графе.
23. Используя нерекурсивную процедуру поиска в глубину, найти кратчайший путь из города A в город B.
24. Используя нерекурсивную процедуру поиска в глубину, найти вершину графа, максимально удаленную от заданной.
25. Используя нерекурсивную процедуру поиска в глубину, найти города, расположенные от города B на расстоянии, большем T.
26. Используя алгоритм поиска в ширину, определить количество шагов, необходимых для нахождения кратчайшего пути из A в B.
27. Для заданного графа найти все вершины, максимально удаленные от вершины A.

28. В заданном графе найти путь между вершинами A и B, проходящий через наименьшее количество вершин графа.
29. Выдать минимальный каркас графа в виде списка входящих в него ребер, используя алгоритм Прима.
30. Выдать минимальный каркас графа в виде списка входящих в него ребер, используя алгоритм Крускала.
31. В графе найти вершину, наиболее удаленную от заданной.
32. Проверить существует ли в данном графе эйлеров путь.
33. Проверить существует ли в данном графе гамильтонов путь.
34. Найти минимальное (по количеству ребер) подмножество ребер, удаление которых превращает заданный связный граф в несвязный.
35. Найти в заданном графе цикл максимальной длины.
36. Проверить правильность утверждения: сумма степеней всех вершин графа равна удвоенному числу ребер.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Дайте определения понятию графа.
2. Приведите примеры разных видов графов.
3. Опишите представление графов в памяти компьютера.
4. Охарактеризуйте отличия матричного представления ориентированных и неориентированных графов?
5. Приведите модель преобразования неориентированного дерево в ориентированное?
6. Перечислите операции над графами.
7. Объясните алгоритмы поиска «в глубину» и «в ширину» в графе.
8. Расскажите о способах обхода графов.
9. Охарактеризуйте различия Эйлерова и Гамильтонова графов.
10. Объясните алгоритм Прима/Крускала.

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 4 занятия (8 академических часов: 7 часов на выполнение и сдачу практического задания и 1 час на подготовку отчета).

Номер варианта студента назначается индивидуально преподавателем.

В отчете должен быть представлен листинг программы, а также снимки экрана результата его работы. Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы, результаты выполнения работы, выводы.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Алексеев В.Е. Графы и алгоритмы. Структуры данных. Модели вычислений [Электронный ресурс]/ В.Е. Алексеев, В.А. Таланов. — Электрон. текстовые данные. — М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 153 с. — Режим доступа: http://www.iprbookshop.ru/52186.html

2. Вирт Никлаус. Алгоритмы и структуры данных [Электронный ресурс]/ Никлаус Вирт— Электрон. текстовые данные. — Саратов: Профобразование, 2017. — 272 с.— Режим доступа: http://www.iprbookshop.ru/63821.html

3. Самуйлов С.В. Алгоритмы и структуры обработки данных [Электронный ресурс]: учебное пособие/ С.В. Самуйлов. — Электрон. текстовые данные. — Саратов: Вузовское образование, 2016. — 132 с.— Режим доступа: http://www.iprbookshop.ru/47275.html

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

4. Костюкова Н.И. Графы и их применение [Электронный ресурс]/ Н.И. Костюкова. — Электрон. текстовые данные. — М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 147 с. — Режим доступа: http://www.iprbookshop.ru/52185.html

5. Сундукова Т.О. Структуры и алгоритмы компьютерной обработки данных [Электронный ресурс]/ Т.О. Сундукова, Г.В. Ваныкина. — Электрон. текстовые данные. — М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 749 с.— Режим доступа: http://www.iprbookshop.ru/57384.html

Электронные ресурсы:

6. Научная электронная библиотека http://elibrary.ru
7. Электронно-библиотечная система «ЛАНЬ» http://e.lanbook.com
8. Электронно-библиотечная система «IPRbooks» http://www.iprbookshop.ru
9. Электронно-библиотечная система «Юрайт» http://www.biblio-online.ru
10. Электронно-библиотечная система «Университетская библиотека ONLINE» http://www.biblioclub.ru