

Министерство образования и науки Российской Федерации
Калужский филиал федерального государственного бюджетного
образовательного учреждения высшего профессионального
образования «Московский государственный технический университет
имени Н.Э. Баумана»
(КФ МГТУ им. Н.Э. Баумана)

Н.И. Пчелинцева, И.К. Белова

РЕАЛИЗАЦИЯ ОСНОВНЫХ АЛГОРИТМОВ РАБОТЫ С ГРАФАМИ

Учебное пособие по курсу «Типы и структуры данных»

Калуга – 2016


Данные методические указания издаются в соответствии с учебным планом кафедры «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» для направления подготовки 09.03.04 «Программная инженерия» КФ МГТУ им. Н.Э. Баумана.

Указания рассмотрены и одобрены:

Кафедрой «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» (ФН1-КФ) 19 мая 2016 протокол № 9

Зав кафедрой ФН1-КФ  Б.М. Логинов


Методической комиссией ФНК 23.05.2016 протокол № 4

Председатель методической комиссии ФНК  К.Л. Анфилов

Методической комиссией Калужского филиала 31.05.16 протокол № 3

Председатель методической комиссии  О.Л. Перерва

Рецензент:

к. т. н., доцент кафедры «Компьютерные системы и сети» (ЭИУ2-КФ)  Е.О. Дерягина

Автор
к. т. н., доцент кафедры ФН1-КФ  Н.И. Пчелинцева

к. ф.-м. н., доцент кафедры ФН1-КФ  И.К. Белова

Аннотация

Методическое пособие по курсу «Типы и структуры данных» содержит основные сведения о графах, часто используемых для создания программных и системных комплексов обработки информации. Рассматриваются способы хранения графов в памяти компьютера, а также основные алгоритмы, применяемые к графам, реализованные на языке программирования C++.

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

© Калужский филиал МГТУ им. Н.Э. Баумана, 2016 г.

© Н.И. Пчелинцева, 2016 г.

© И.К. Белова, 2016 г.

ОГЛАВЛЕНИЕ

	Стр.
ВВЕДЕНИЕ	5
ОСНОВНЫЕ СВЕДЕНИЯ О ГРАФАХ.....	6
ПРЕДСТАВЛЕНИЕ ГРАФОВ В ПАМЯТИ КОМПЬЮТЕРА	8
ОБХОДЫ ГРАФОВ.....	12
Алгоритм поиска в глубину	12
Алгоритм поиска в ширину	15
НАХОЖДЕНИЕ КРАТЧАЙШЕГО ПУТИ	16
Алгоритм Дейкстры.....	16
Алгоритм Беллмана-Форда.....	19
Алгоритм Флойда-Уоршелла.....	22
ОСНОВНЫЕ ДЕРЕВЬЯ ГРАФОВ	25
Алгоритм Прима	26
Алгоритм Крускала.....	29
ЛИТЕРАТУРА	32

ВВЕДЕНИЕ

Настоящее методическое пособие подготовлено в соответствии с учебным планом кафедры «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» для направления подготовки 09.03.04 «Программная инженерия» КФ МГТУ им. Н.Э. Баумана.

Пособие посвящено теории графов и состоит из пяти разделов. В первом разделе даны основные понятия и определения теории графов, рассмотрены виды графов. Второй раздел посвящен способам представления графов в памяти компьютера. Рассматриваются основные структуры данных, с помощью которых можно описать граф в памяти компьютера. Третий раздел посвящен основной операции работы с графом – перебору всех вершин графа с помощью классических алгоритмов поиска в ширину и глубину. В четвертом разделе разбираются фундаментальные алгоритмы поиска кратчайших путей. В Важнейший вид графов – деревья – рассмотрен в пятом разделе. Разобраны алгоритмы нахождения остовных деревьев минимальной стоимости.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия». Пособие может быть полезным студенту при подготовке к лекциям, практическим занятиям, рубежным контролям и итоговому зачету.

Основные сведения о графах

Граф – это 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 инцидентны 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.

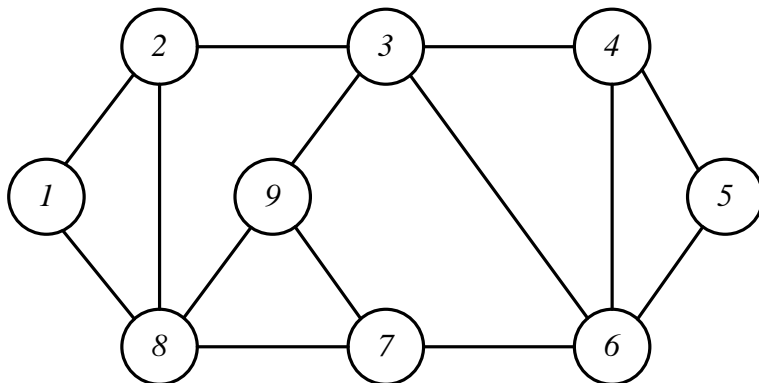


Рис. 1. Неориентированный граф

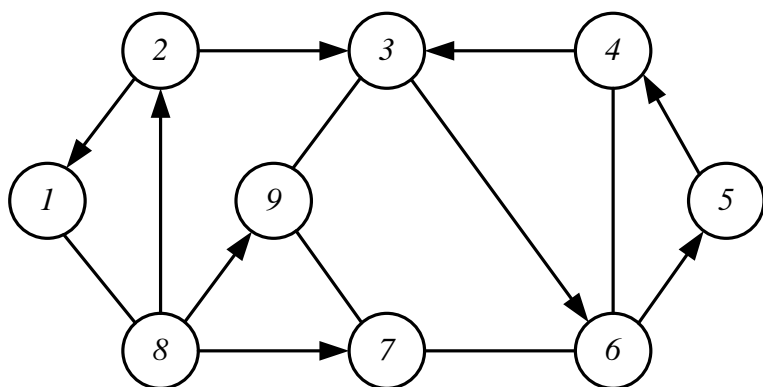


Рис. 2. Смешанный граф

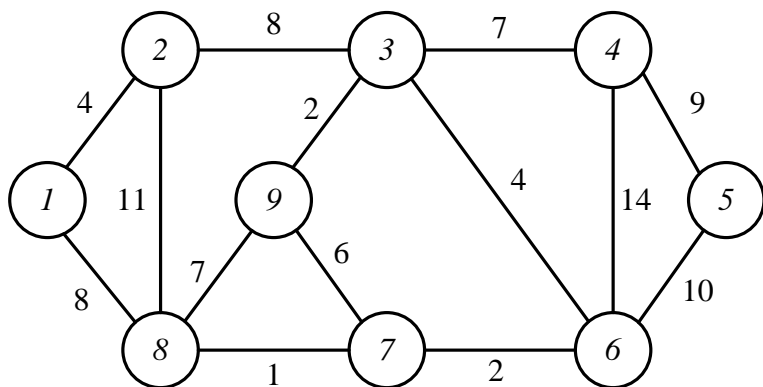


Рис. 3. Взвешенный граф

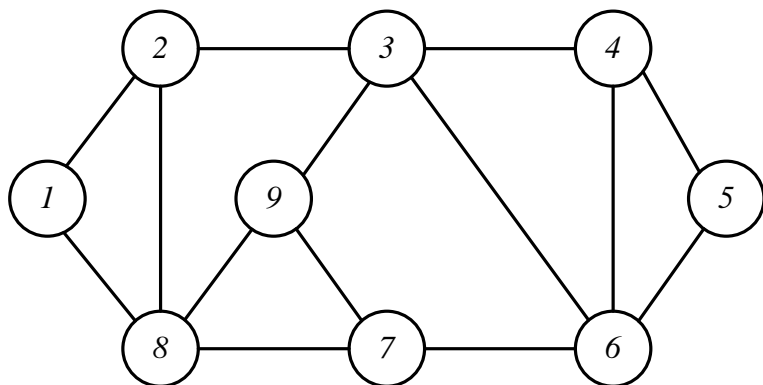


Рис. 4. Мультиграф

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

Вот еще несколько определений 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 рёбра 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 рёбер (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 (рёбер), в которой каждый элемент инцидентен предыдущему и последующему. Может рассматриваться как частный случай маршрута.

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

Цикл – замкнутая цепь. Для орграфов цикл называется контуром.

Эйлеров граф – это граф, в котором существует цикл, содержащий все рёбра 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 рёбра графа (вершины могут повторяться).

Представление графов в памяти компьютера

Графы в памяти могут представляться различными способами. Один из видов 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.

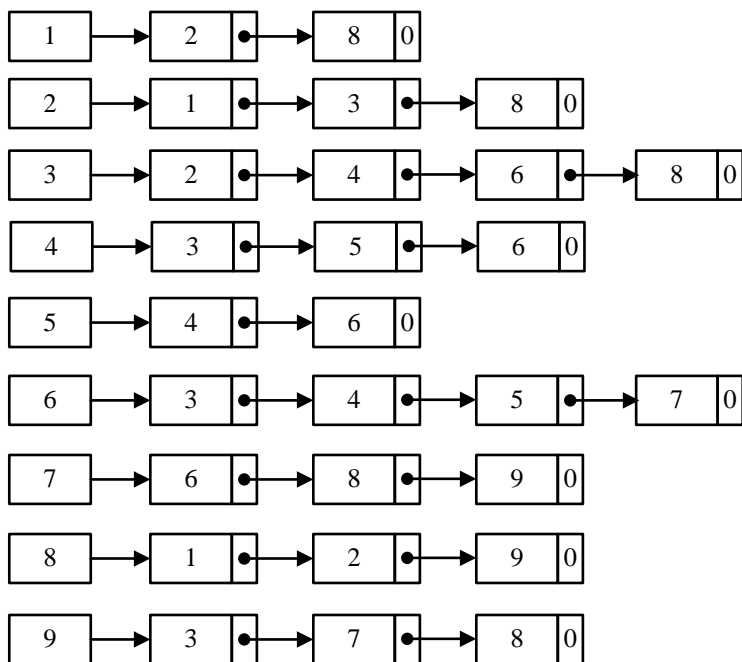


Рис. 5. Список смежностей

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

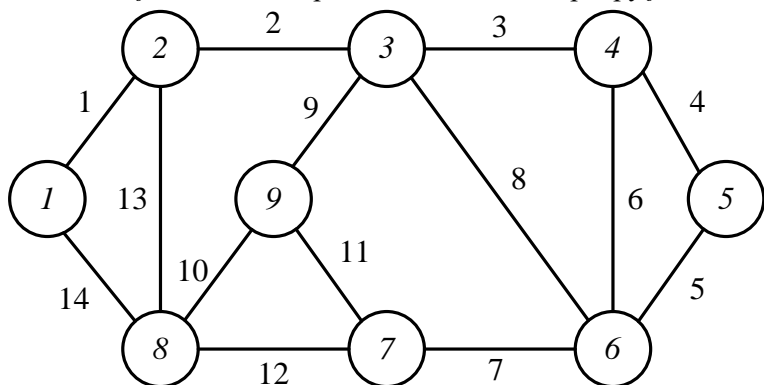


Рис. 6. Неориентированный граф

Например, для графа с рисунка 6 матрица инцидентности 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];
            }
    }
}
```

```

    int & operator () (int _i, int _j)//обращение к эле-
менту матрицы
    {
        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 об-

работана). 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).

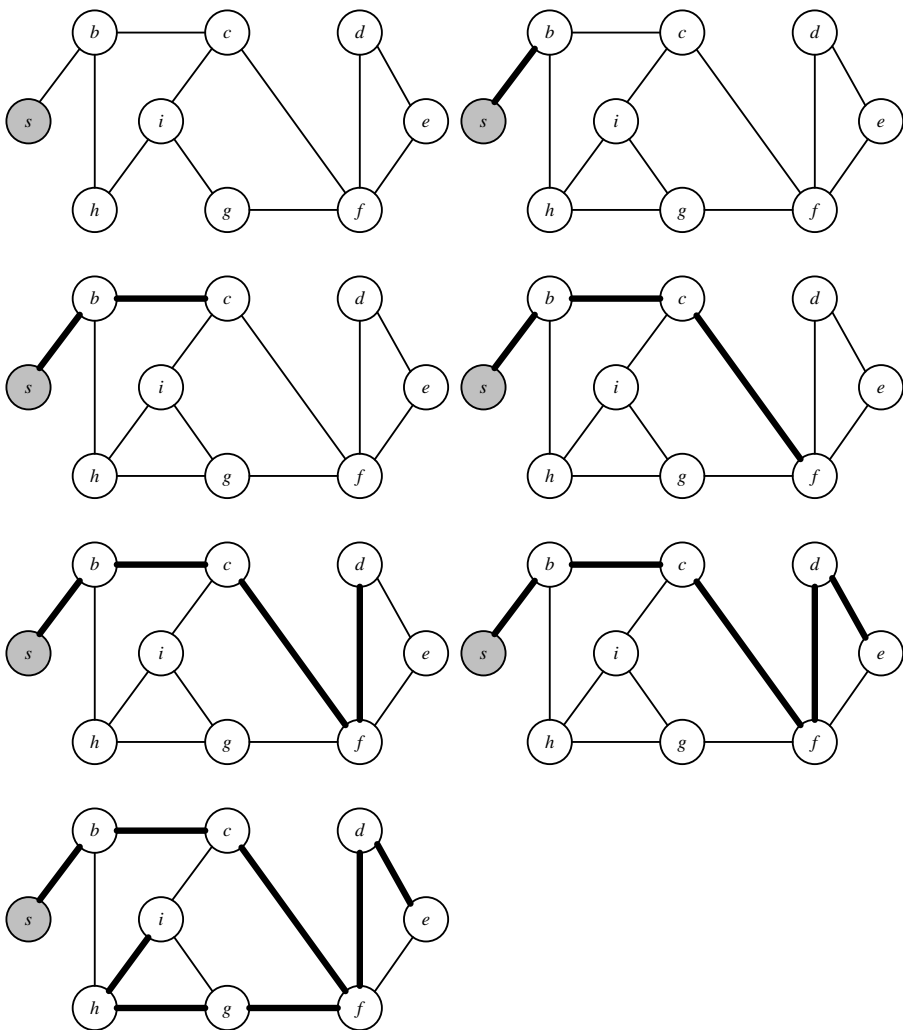


Рис. 7. Выполнение алгоритма DFS

Реализация алгоритма поиска в глубину 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;
```


Реализация алгоритма обхода в ширину 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);
}
```

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 путей из одной выбранной вершины ко всем остальным вершинам при любых весах ребер, в том числе и отрицательных.

Для поиска кратчайших путей между всеми вершинами используется алгоритм Флойда-Уоршалла.

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.

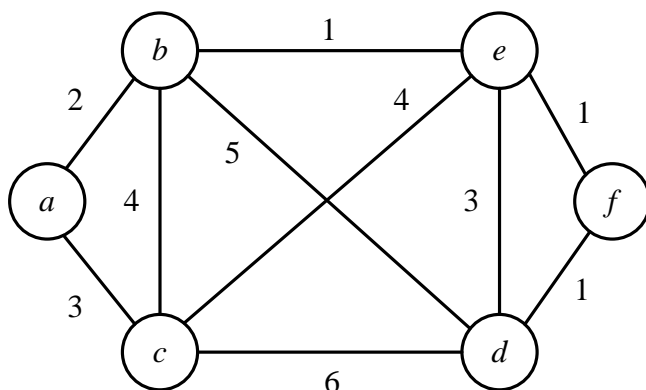
Каждой вершине приписывается вес – это вес пути от начальной вершины до данной. Также каждая вершина может быть выделена. Если вершина выделена, то путь от нее до начальной вершины кратчайший, если нет – то временный. Обходя граф, алгоритм считает для каждой вершины маршрут, и, если он оказывается кратчайшим, 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 $G=(V, E)$, $n=|V|$, 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.

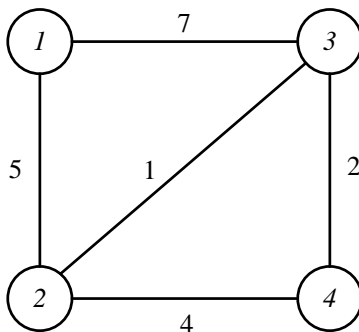


Рис. 10. Пример графа

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

Перед k -ой итерацией ($k = 1 \dots n$) считается, что в матрице расстояний $w[i][j]$ сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $[1, \dots, k-1]$ (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 $w[i][j]$ 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 $h[i][j]$, 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 $w[i][j]$ 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: $1 \rightarrow 4 = 3$, $1 \rightarrow 3 = 2$, $1 \rightarrow 2 = 2$. Следовательно, путь будет следующим: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

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 рёбер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим рёбрам, и в нём 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).

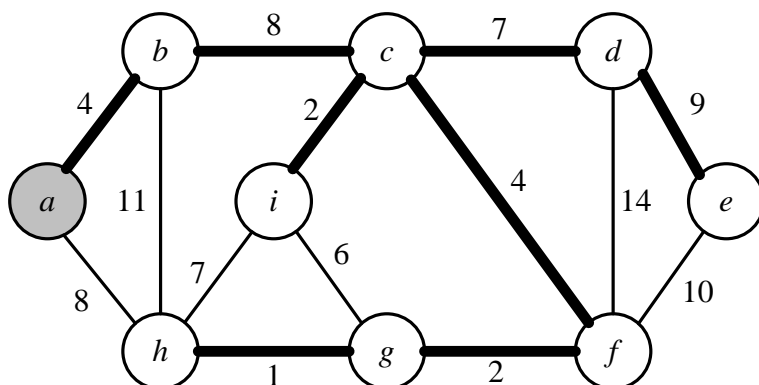


Рис. 11. MST-дерево графа

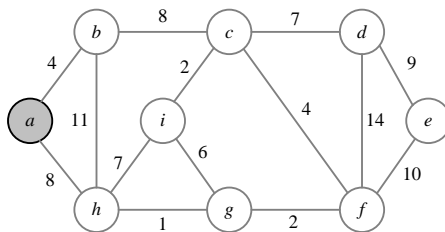
Задача поиска минимального остовного дерева в произвольном взвешенном неориентированном графе применяется во многих ситуациях. Наиболее часто для построения остовых деревьев минимальной стоимости используют два классических алгоритма: алгоритм Крускала и алгоритм Прима. Рассмотрим их подробнее.

Алгоритм Прима

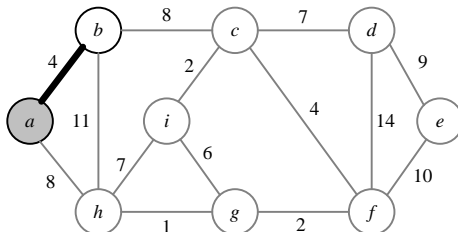
Алгоритм Прима обладает тем свойством, что ребра во множестве E всегда образуют единое дерево. Дерево начинается с произвольной корневой вершины a и растет до тех пор, пока не охватит все вершины в V . На каждом шаге к дереву добавляется легкое ребро, соединяющее дерево и отдельную вершину из оставшейся 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:

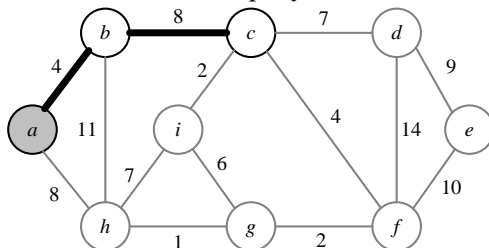
1. В качестве входных данных алгоритму передаются связный граф G и корень a минимального остовного дерева.



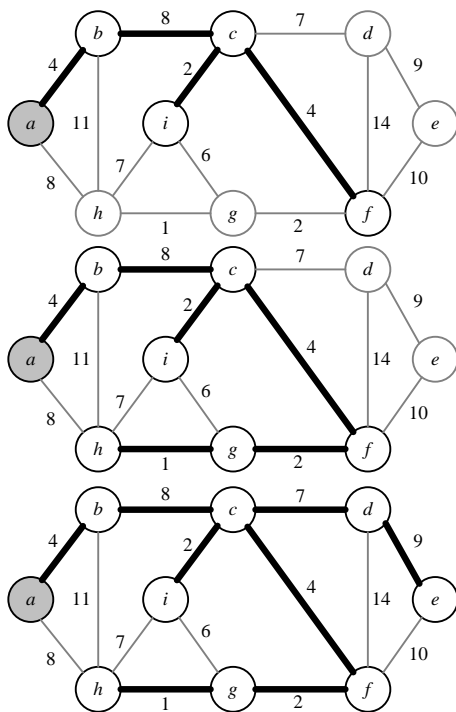
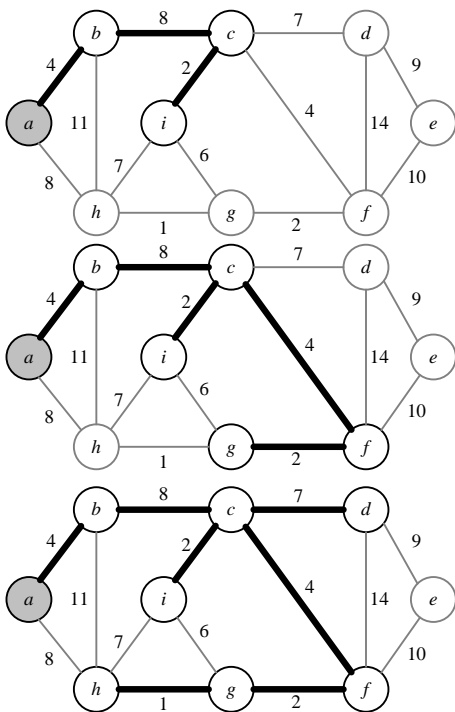
2. Ребро с весом 4 является минимальным и добавляется к дереву.



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



4. 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;

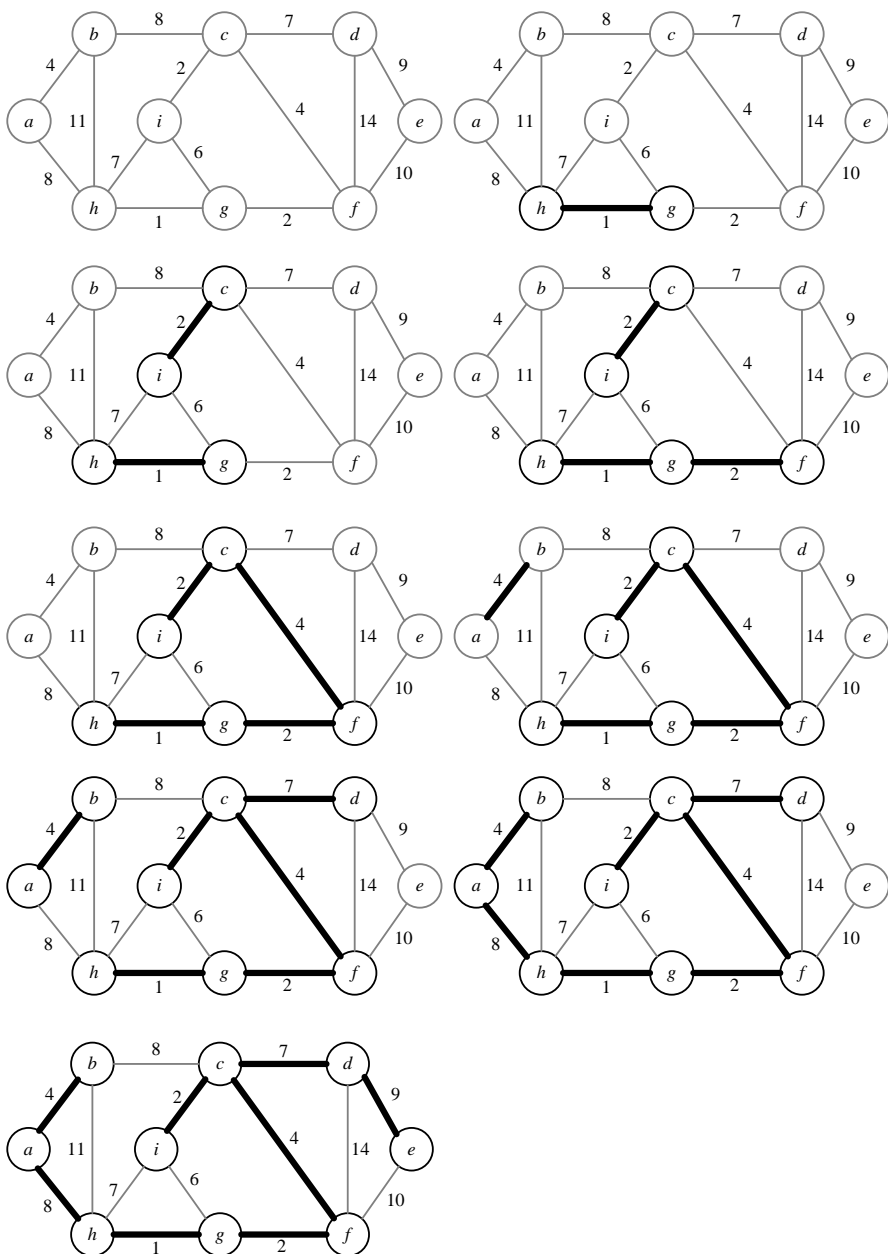
    for (int to=0; to<n; ++to)
        if (g[v][to] < min_e[to]) {
            min_e[to] = g[v][to];
            sel_e[to] = v;
        }
}

```

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:



Алгоритм Крускала на языке C++:

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

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;
    }
}
```

ЛИТЕРАТУРА

1. Никлаус Вирт Алгоритмы и структуры данных. Новая версия для Оберона [Электронный ресурс]/ Никлаус Вирт— Электрон. текстовые данные.— М.: ДМК Пресс, 2010.— 272 с.— Режим доступа: <http://www.iprbookshop.ru/7965>.— ЭБС «IPRbooks», по паролю
2. Сундукова Т.О. Структуры и алгоритмы компьютерной обработки данных [Электронный ресурс]/ Сундукова Т.О., Ваныкина Г.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2011.— 475 с.— Режим доступа: <http://www.iprbookshop.ru/16736>.— ЭБС «IPRbooks», по паролю
3. Роберт Л. Круз Структуры данных и проектирование программ [Электронный ресурс]/ Роберт Л. Круз— Электрон. текстовые данные.— М.: БИНОМ. Лаборатория знаний, 2014.— 766 с.— Режим доступа: <http://www.iprbookshop.ru/37101>.— ЭБС «IPRbooks», по паролю
4. Кнут Д. Э. Искусство программирования. Т. 1: пер. с англ. / Дональд Э. Кнут; под общ. ред. Ю. В. Козаченко. - М. [и др.], 2007. - 712 с.

**Наталья Ибрагимовна Пчелинцева
Ирина Константиновна Белова**

**РЕАЛИЗАЦИЯ ОСНОВНЫХ АЛГОРИТМОВ
РАБОТЫ С ГРАФАМИ**

Методическое пособие