Структура данных Граф

Структура данных Граф может быть использована для представления данных, моделей процессов.

Многие приложения используют не только набор элементов данных, но также и набор соединений между парами этих элементов.

Отношения, которые вытекают из этих соединений, приводят к вопросам:

- Существует ли путь от одного такого элемента к другому, вдоль этого соединения?
- В какие еще элементы можно перейти из заданного элемента?
- Какой путь от одного заданного элемента к другому наилучший?

Примеры алгоритмов использующих граф как структуру данных

- При выборе пути от одного населенного пункта к другому необходимо реализовать географическую карту местности.

- При поиске в Интернет необходимой информации мы сталкиваемся в документах со ссылками на другие документы

- Составление расписания (например, занятий) – производственный процесс требует решения задач при этом должно учитываться множество ограничений, по условиям которых решение одной задачи не может быть начато, пока не будет завершено решение другой задачи

Рассматривая Граф как структуру данных необходимо определить вид связи (отношения) между её элементами.

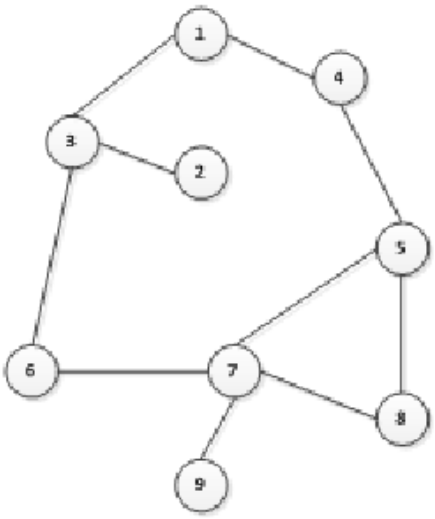
В графе между элементами устанавливается отношение соединения пар элементов.

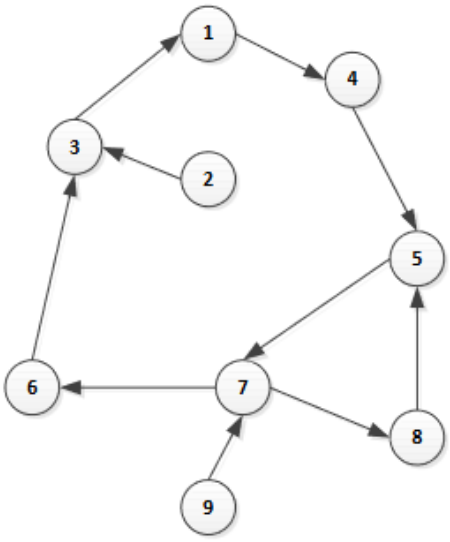
Если для линейной структуры мы определяем отношение 1 к 1, а в иерархических структурах отношение 1 к многим (М) - у каждого потомка только один предок, то в графе отношения между элементами Многие к Многим (М к М).

Если при определении данных задачи вы встречаете такие отношения, то реализовать эти данные следует через граф.

Граф является математическим объектом, он исследован, для него определен математический аппарат, а значит и задачи по обработке таких данных существует.

Определение
Граф - Конечное не пустое множество вершин (V) и конечное множество упорядоченных или неупорядоченных пар вершин (ребер).
Обозначим G={{V},{E}}.





Основные понятия. Изображение графа: вершины - множество точек, соединенных линиями, которые определяют ребра графа

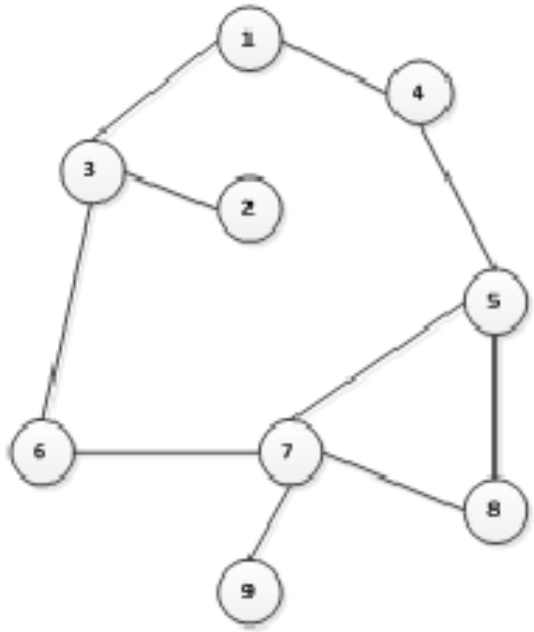
Ребро – неупорядоченная пара вершин.
Обозначение ребра (u,v) – ребро соединяет вершины u и v.

Дуга–упорядоченная пара вершин.

Неориентированный граф - содержит только неупорядоченные ребра.

Ориентированный граф – содержит только дуги.
Вершины, соединенные ребром называются **смежными**.
Ребра, имеющие общую вершину, также называются **смежными**.
Ребро и любая из вершин называются **инцидентными**.

Степень вершины – это число ребер, инцидентных этой вершине.
Пути от вершины 1 к вершине 8:
1 4 5 8
1 3 6 7 8
1 3 6 7 5 8

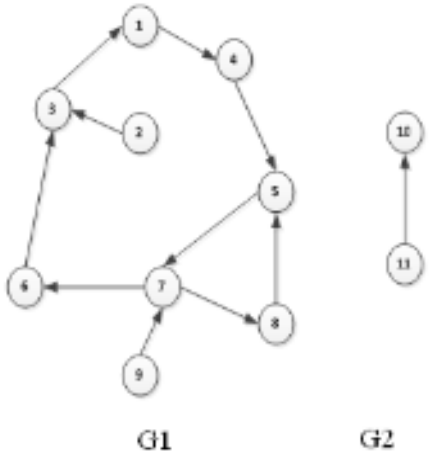


Полный граф - граф у которого каждая вершина смежная со всеми остальными вершинами.

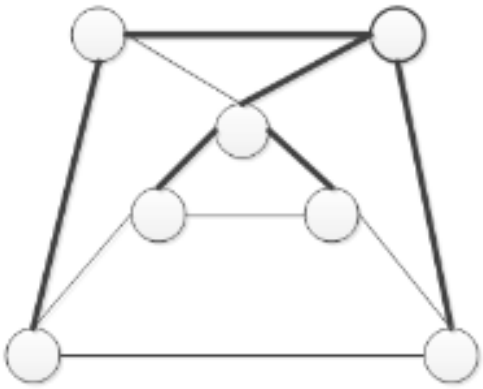
Связный граф – это граф, в котором существует путь из каждой вершины в любую другую вершину графа.
Граф, состоящий из N вершин содержит не более N(N-1) /2 ребер.

Взвешенный граф – это граф, ребра которого имеют вес – некоторое значение, которое определяет расстояние между вершинами или время и т.д.

Несвязный граф состоит из некоторого множества связных компонентов, которые представляют собой максимальные связные подграфы.



Остовное дерево связного графа – это подграф, который одержит все вершины этого графа и представляет единое дерево.
Ациклический связный граф называется **деревом**.
Множество деревьев называется **лесом**.



Расстояние между вершинами - это длина простого пути наименьшей длины.

Для фиксированной вершины графа расстояние до наиболее удаленной от нее вершины называют **эксцентриситетом** (максимальным удалением) вершины.

Диаметром графа называют число d , равное расстоянию между наиболее удаленными друг от друга вершинами графа.

Цикл (контур) – путь, начинающийся и заканчивающийся в одной и той же вершине.

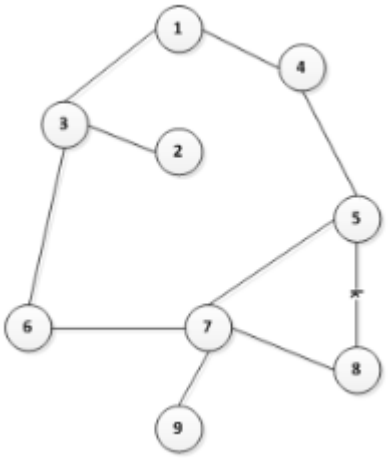
Простой цикл – соответствующий ему путь простой.

Ациклический граф – это граф без циклов.

Представление графа в памяти

Матрица смежности - двумерный массив размерности V×V.
где E –количество ребер; V –количество вершин.

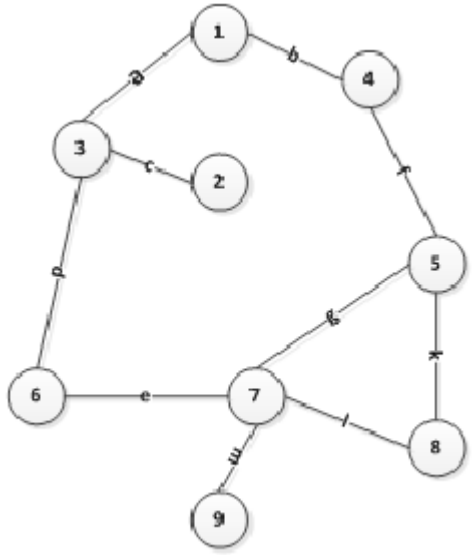
	1	2	3	4	5	6	7	8	9
1	0	0	1	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	1	1	0	0	0	1	0	0	0
4	1	0	0	0	1	0	0	0	0
5	0	0	0	1	0	0	1	1	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	1	1	0	1	1
8	0	0	0	0	1	0	1	0	0
9	0	0	0	0	0	0	1	0	0



Представление графа в памяти

Матрица инцидентности – хранит связи между инцидентными элементами графа (ребро(дуга) и вершина).

	a	b	c	d	e	f	g	k	l	m
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	1	0	1	1	0	0	0	0	0	0
4	0	1	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	1	1	0	0
6	0	0	0	1	1	0	0	0	0	0
7	0	0	0	0	1	0	1	0	1	1
8	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	0	1



Представление ориентированного графа в памяти

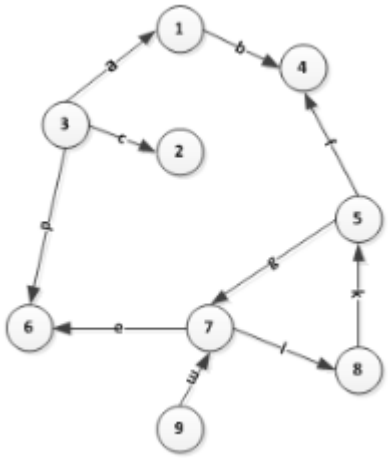
Ориентированный граф

1 –вершина инцидентна ребру, и является его началом

0 –вершина не инцидентна ребру

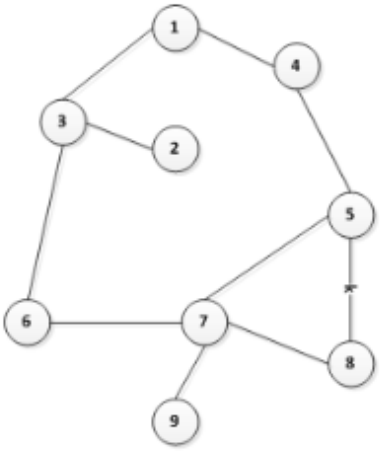
-1 –вершина инцидентна ребру, и является его концом

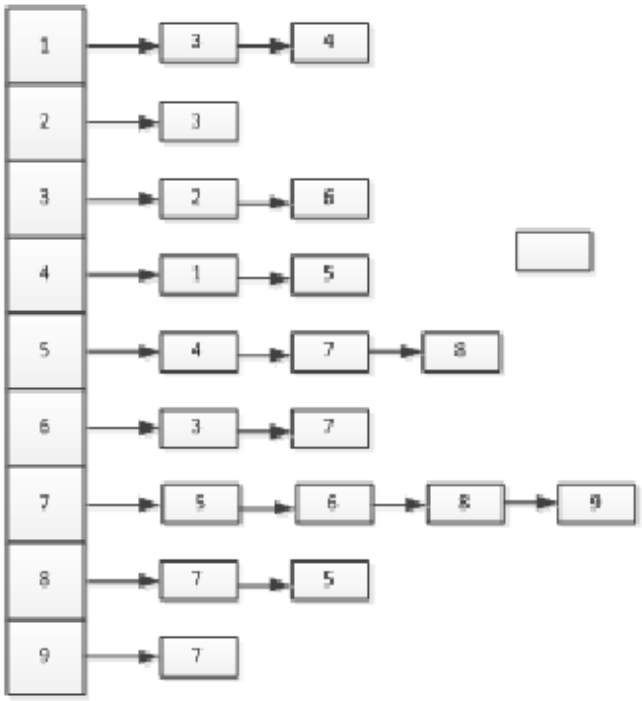
	a	b	c	d	e	f	g	k	l	m
1	-1	1	0	0	0	0	0	0	0	0
2	0	0	-1	0	0	0	0	0	0	0
3	1	0	1	1	0	0	0	0	0	0
4	0	-1	0	0	0	-1	0	0	0	0
5	0	0	0	0	0	1	1	-1	0	0
6	0	0	0	-1	-1	0	0	0	0	0
7	0	0	0	0	1	0	-1	0	1	-1
8	0	0	0	0	0	0	0	1	-1	0
9	0	0	0	0	0	0	0	0	0	-1



Представление графа в памяти

Списки смежных вершин – связанные списки, содержащие список вершин, смежных заданной вершине.



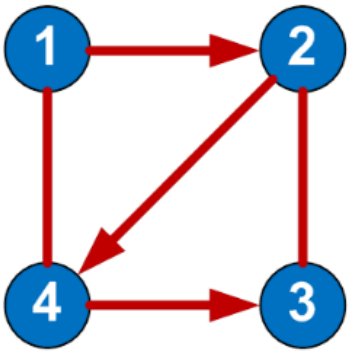


Представление графа в памяти

перечень ребер

В списке рёбер в каждой строке записываются две смежные вершины и вес соединяющего их ребра (для взвешенного графа). Количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.

	Начало	Конец	Вес
1	1	2	
2	1	4	
3	2	3	
4	2	4	
5	3	2	
6	4	1	
7	4	3	



Абстрактный тип данных
АТД Graph
{
Данные:
Множество вершин типа Tvert и ребер.
Количество ребер E и вершин V.
typedef struct {int v; int w;}Edge;
typedef struct graph *Graph;

Операции:
//Создание ребра
Edge EDGE(int u,int v);

//Инициализация графа из V вершин
Graph GRAPHInit(int V);

//Вставка ребра в граф
void GRAPHInsertE(Graph ,Edge);

//Удаление ребра
void GRAPHremovE(Graph ,Edge);

//Обработка значений в вершине на примере заполнения массива ребрами графа и подсчитывающей количество ребер
int GRAPHhedges(Edge [],Graph);

//Копирование графа
Graph GRAPHcopy(Graph);

//удаляет граф
void GRAPHdestroy(Graph);

//Обход графа в глубину с заданной вершины
void dfsR(Graph,int);

//Обход графа в ширину с вершины по инцидентным ей ребрам
void bfs(Graph,Edge);
}

```
struct Edge
{int v; int w;};
struct graph
{int V;
int E;
int **Matr;
};
typedef graph *Graph;
//создание ребра
Edge EDGE(int,int);
Graph GRAPHInit(int V)//создание матрицы смежности
{ Graph g=new graph;
   g->V=V;
   g->Matr=new int *[V];
   for(int v=0;v<V; v++)
        g->Matr[v]=new int[V];
   for (int v=0;v<V;v++)
       for(int w=0;w<V;w++)
            g->Matr[v][w]=0
    return g;
}
```

Вставка ребра в матрицу смежности ***неориентированного*** графа

```
void GRAPHInsert(Graph G,Edge e)
{   int  v=e.v,w=e.w
    if(G->Matr[v][w]==0) G->E++;
    G->Matr[v][w]=1;
    G->Matr[w][v]=1;
}
const int maxV=10;
void main()
{   Edge e;
    Graph G;
    G=GRAPHInit(maxV);
    int v;
    e.v=1;e.w=4;GRAPHInsertE(G,e);
    e.v=4;e.w=5;GRAPHInsertE(G,e);
    e.v=5;e.w=8;GRAPHInsertE(G,e);
    e.v=5;e.w=7;GRAPHInsertE(G,e);
    e.v=8;e.w=7;GRAPHInsertE(G,e);
    e.v=9;e.w=7;GRAPHInsertE(G,e);
    e.v=7;e.w=6;GRAPHInsertE(G,e);
    e.v=6;e.w=3;GRAPHInsertE(G,e);

    e.v=3;e.w=2;GRAPHInsertE(G,e);
    e.v=3;e.w=1;GRAPHInsertE(G,e);
    e.v=3;e.w=2;GRAPHremovE(G,e);
    Pmatr(G);

    v=1;
    dfsR(G,v);
    e.v=1;e.w=3;
    bfs(G,e);
    system("PAUSE");
}
```

Пути в графе

Достижимость вершин в ориентированном графе. Во многих задачах требуется определение путей в графах:

между двумя вершинами, всех путей
между двумя вершинами, экстремальных путей,
путей от одной вершины ко всем другим,
определение дуг, составляющих путь.

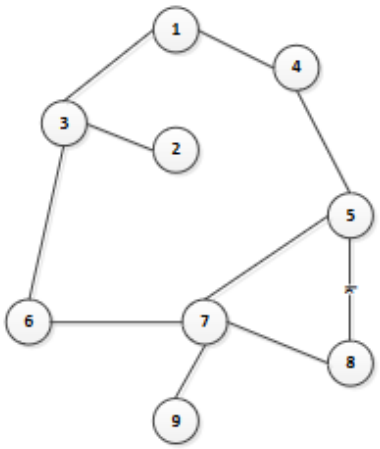
Путем ориентированного графа называется последовательность дуг, в которой конечная вершина всякой дуги, отличной от последней, является начальной вершиной следующей дуги.

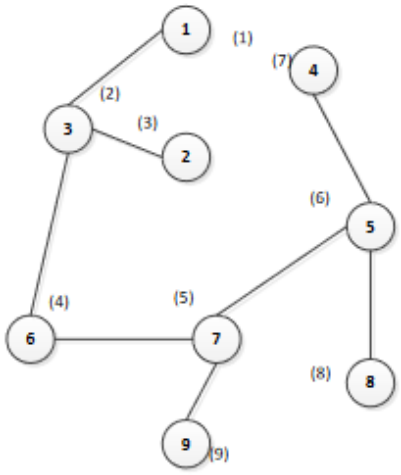
Простой путь – это путь, в котором каждая дуга используется не более одного раза.

Вершина w достижима из вершины v, если существует путь из вершины v в вершину w.

Поиск в графе

Поиск в глубину не ориентированного графа





	1	2	3	4	5	6	7	8	9
1	0	0	1	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	1	1	0	0	0	1	0	0	0
4	1	0	0	0	1	0	0	0	0
5	0	0	0	1	0	0	1	1	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	1	1	0	1	1
8	0	0	0	0	1	0	1	0	0
9	0	0	0	0	0	0	1	0	0

-1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9

1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9

При реализации алгоритма для запоминания уже посещенных вершин используется дополнительная структура, часто массив.

Алгоритм обхода (поиска) в глубину не ориентированного графа

Поиск начинается с некоторой фиксированной вершины v.

Рассматривается вершина w смежная с v. Процесс повторяется с вершиной w.

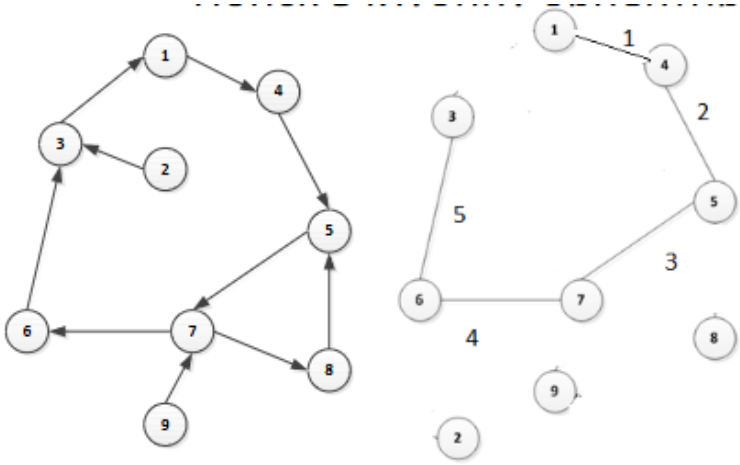
Если на очередном шаге при работе с вершиной x, нет вершин смежных с x, то возвращаемся от x к вершине, которая была до нее.

В случае, когда это вершина v, то процесс завершается.

Для контроля за уже посещенной вершиной надо использовать массив L признаков из V элементов. Заполнить его значением -1. Устанавливать в 1 элемент с индексом v, при рассмотрении вершины v.

```
Dfs(G,v){
W- смежная вершина
L[v]<-1 –признак пройденной вершины
For w<-1 to V do
If (L[w]=-1
Dfs(w);
Od
}
//инициализация массива не посещенных вершин
For v<-1 to V do L[v]<- (-1) od
// обход графа полный с учетом не связного графа
For v<-1 to V do
if L[v]=1 Dfs (v)
od
```

Поиск в глубину ориентированного графа



	1	2	3	4	5	6	7	8	9
1	0	0	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	1	0	0	0	0	0	0
7	0	0	0	0	0	1	0	1	0
8	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	1	0	0

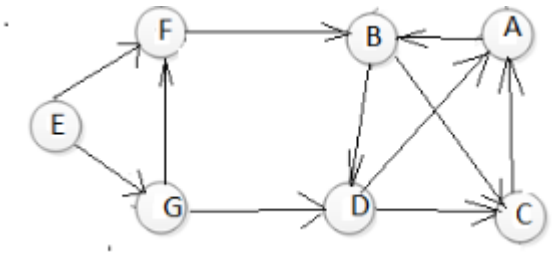
-1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9

1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8	9

Анализ поиска в глубину ориентированного графа

Все вызовы Dfs для полного обхода графа с V вершинами и E дугами, если E>=V требуют времени O(E). Так как нет вершин, для которых Dfs вызывалась бы больше одного раза, поскольку рассматриваемая вершина помечена как посещенная еще до следующего вызова Dfs и никогда не вызывается для вершин, помеченных как посещенная.

В представленном на рисунке графе будет поиск в одном подграфе ABDC, затем обход продолжиться с не посещенной вершины E



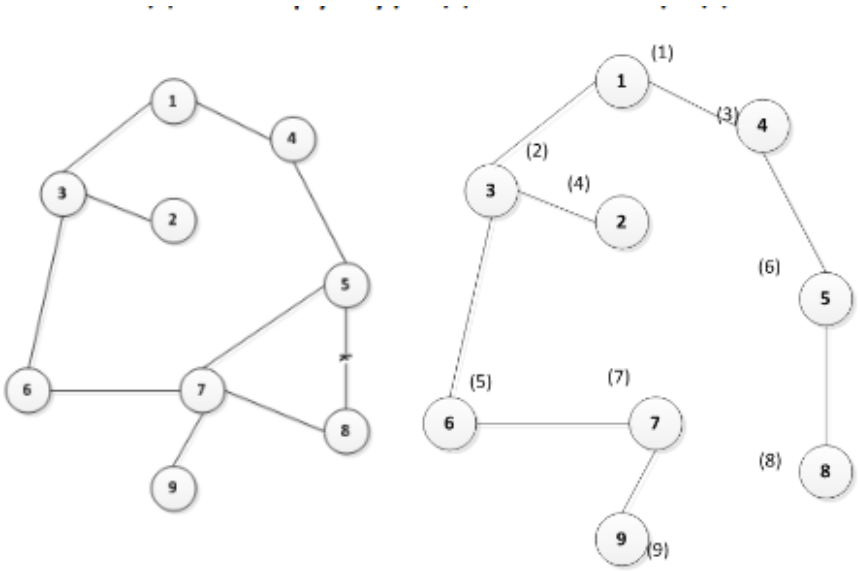
обход ориентированного графа в глубину

```
void dfsR(Graph G,int v)//
{int *L=new int [G->V];
for (int v1=1;v1<G->V;v1++)// список не посещенных вершин
   L[v1]=-1;
for(int v1=v;v1<G->V;v1++)//обход графа
   if(L[v1]==-1)
   {cout<<"\n"<<v1;
     dfs(G,v1,G->V,L);
    }
}
void dfs(Graph G, int v, int V, int L[])
{int w;
L[v]=1;
for(w=1;w<V;w++)
  if (G->Matr[v][w]==1)
     if(L[w]==-1)
        { cout<<"->"<<w;
          dfs(G,w,V,L);
         }
}
```

Поиск в ширину в неориентированном графе

Просмотр вершины сводится к просмотру инцидентных вершин и помещению всех ребер , ведущих на не посещенную вершину, в очередь вершин для просмотра.

Рассматриваются все вершины, связанные с текущей. Принцип выбора следующей вершины – выбирается та, которая была раньше рассмотрена. Для реализации данного принципа необходима структура данных очередь.



	1	2	3	4	5	6	7	8	9
1	0	0	1	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	1	1	0	0	0	1	0	0	0
4	1	0	0	0	1	0	0	0	0
5	0	0	0	1	0	0	1	1	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	1	1	0	1	1
8	0	0	0	0	1	0	1	0	0
9	0	0	0	0	0	0	1	0	0

```
void bfs(Graph G,Edge e)
{
   queue Q(G->V);Edge q;
   int w,numb=0;
   Q.push(e);
   int *L=new int[G->V];
   for(int i=0;i<G->V;i++) L[i]=-1;
   L[e.v]=1;
   cout<<e.v;
   while (Q.Empty())
  {  q=Q.peek();
     L[q.w]=e.v;
     for(int v=1;v<G->V;v++)
           if (G->Matr[e.v][v]==1 && L[v]==-1)
           {    Edge q1; q1.v=e.v; q1.w=v;
                Q.push(q1);
                cout<<'-'<<q1.w;
                L[q1.w]=1;
                //Tree[1][numb]=w;Tree[2][numb]=v;numb++;} }}
```

Кратчайшие пути в графе

Поиск кратчайших путей. Задачи о кратчайших путях относят к фундаментальным задачам комбинаторной оптимизации, так как многие задачи можно свести к отысканию кратчайших путей в графе.

Типы задач о кратчайшем пути между вершинами

- Между двумя заданными
- Между данной и всеми остальными вершинами
- Каждой парой вершин и т.д.

Алгоритм Дейкстры
Определить кратчайший путь между парой заданных вершин. Применяется к ориентированным, не ориентированным графам.

Дано: Граф G{V,E} с N вершинами
S –вершин а источник. Матрица смежности (весов) A[N][N]

Для любых w, v $\in V$ вес дуги A[w][v]>=0. Если дуга от w к v отсутствует, то A[w][v] = ∞(число большее самой длинной дуги).

Результат. Массив кратчайших расстояний D – перечень дуг, через которые проходит кратчайший путь и дуга представлена тройкой значений: начальная вершина, конечная вершина и длина пути от заданной вершины v до конечной вершины дуги.

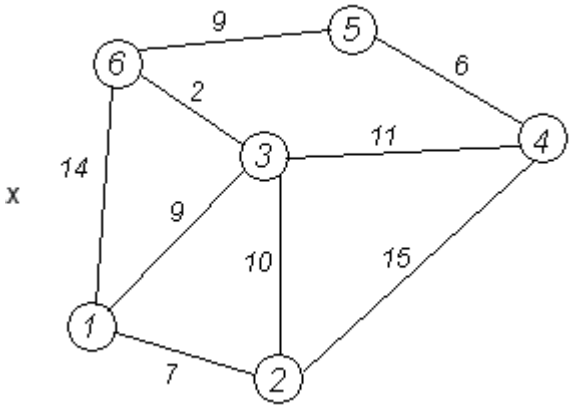
Описание. Формирует множество вершин S, для которых еще не вычислено расстояние и минимальное значение в D. По множеству вершин принадлежащих S считается окончательная оценка для вершины, на которой достигается этот минимум.

Алгоритм

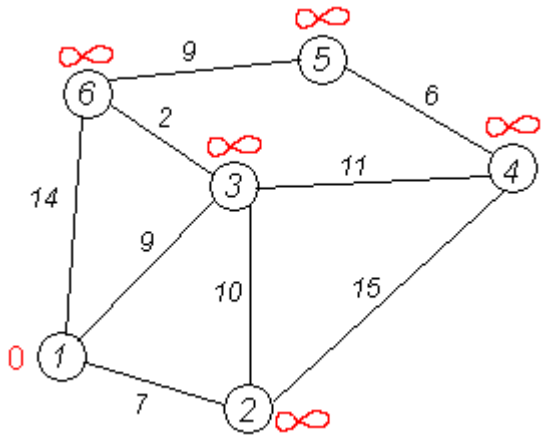
1. Поступает начальная вершина v и конечная вершина w до которой нужно вычислить кратчайшее расстояние.
2. Остальные N-1 вершин включаем в перечень не выбранных вершин.
3. Определяем расстояние до всех остальных не выбранных вершин.
4. Выбираем вершину до которой расстояние наименьшее. Исключаем эту вершину из перечня не выбранных. Дугу перехода, определившую наименьшее расстояние, включаем в перечень дуг кратчайшего пути.
5. Определяем кратчайшее расстояние от выбранной вершины до всех не выбранных вершин с учетом пройденных от начальной точки v расстояний.
6. Если расстояние до некоторой вершины меньше ранее определенного расстояния, то заменяем его на новое меньшее.
7. Выбираем опять наименьшее расстояние и новую вершину исключаем из списка не выбранных. Т.е. повторяем с п.4. пока не достигнем вершины w или не выберем все N вершин при поиске кратчайшего пути о вершины v до всех остальных вершин.

Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке.

Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных. Кружками обозначены вершины, линиями — пути между ними (рёбра графа).

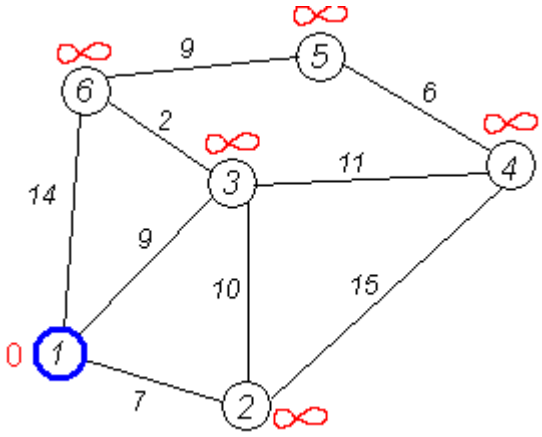


Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.



Первый шаг.

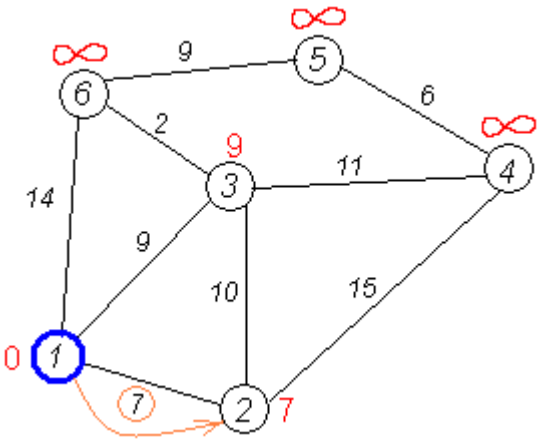
Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6.



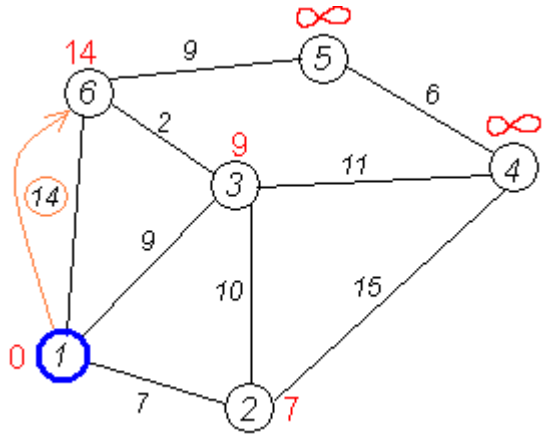
Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё минимальна.

Длина пути в неё через вершину 1 равна сумме значения метки вершины 1 и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7.

Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.



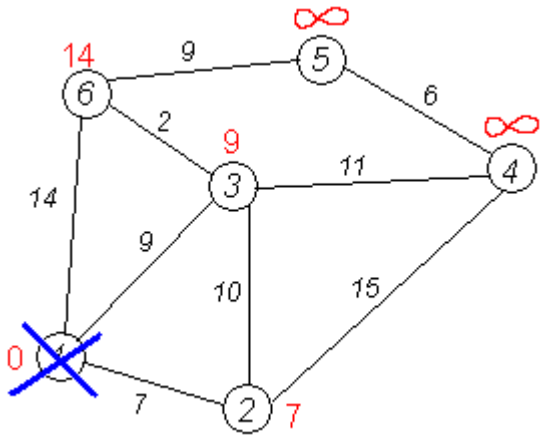
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.



Все соседи вершины 1 проверены.

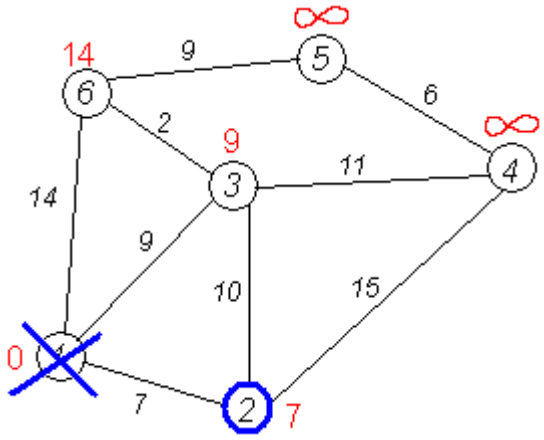
Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит.

Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



Второй шаг.

Снова находим «ближайшую» из не посещённых вершин. Это вершина 2 с меткой 7.

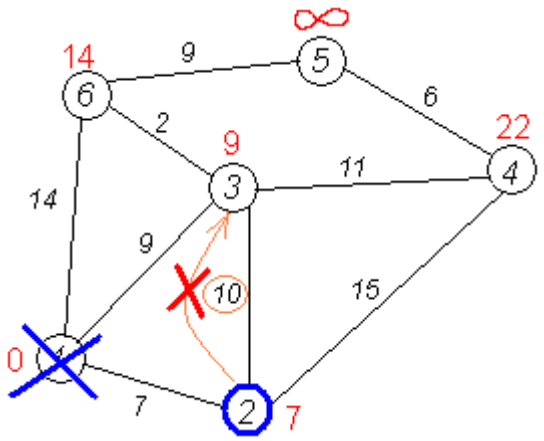


Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

Следующий сосед — вершина 3, так как имеет минимальную метку.

Если идти в неё через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9, а это меньше 17, поэтому метка не меняется.

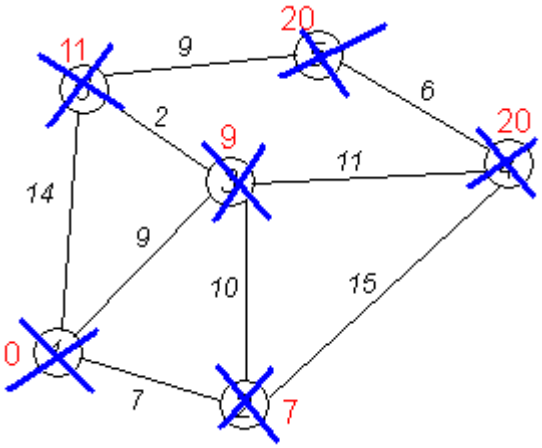


Ещё один сосед вершины 2 — вершина 4.

Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 (7 + 15 = 22).

Поскольку 22< ∞, устанавливаем метку вершины 4 равной 22.

Повторяем шаг алгоритма для оставшихся вершин. Алгоритм заканчивает работу, когда все вершины посещены.



```
 Deikstra (v,w,C, N){
V{1,2,3,4,5}
S={}
For i<-1 to N do
D[i]<-C[1,i]
Od
For i<-2 to N do
P[i]<-1
od
S= {v}
While (V<{})
do
   min=∞
   For i in V-S do
   If D[i]<min
   min=D[i] w=i
   od
   Do
   S=S+{w}
      For v in V-S do
      D[v]=min(D[v], D[w]+C[w,v]
      If(D[v]+C[w,v]<D[v]) P[v]=w
   od
Od
P[2]=1 P[3]=4P[4]=1P[5]=3
```

Алгоритм Флойда
Применяется в задачах, когда надо найти кратчайший путь для каждой пары вершин (w,v). Эту задачу можно решить применяя алгоритм Дейкстры, но возрастает сложность алгоритма. При использовании матрицы смежности сложность будет равна $O(N^3)$. При использовании списка смежных вершин O(e*N*logN).

Дано. Граф G{V,E} представлен взвешенной матрицей смежности C с N вершинами.

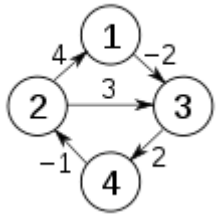
Результат. Матрица кратчайших расстояний A, в которой каждый элемент A[i][j] содержит значение кратчайшего пути между вершинами i и j.

Идея алгоритма Флойда построена на поиске кратчайшего пути между двумя вершинами i и j , проходящими через вершины k меньшие i и j.

1. Сначала матрица C копируется в матрицу A.
2. Если дуга (i,j) отсутствует, то заменяется на бесконечность. Алгоритм выполняется за N проходов.
3. После k-ого прохода A[i][j] содержит значение наименьшей длины путей из вершины i к j, которые не проходят через вершины с номерами большими k, т.е. между вершинами i и j могут быть вершины только с номерами меньшими или равными k.
4. При k-ом проходе элемент A[i][j] вычисляется по формуле Ak[i][j]=min(Ak-1[i][j], Ak-1[i][k]+ Ak-1[i][j])
5. Чтобы получить сам путь используется дополнительно матрица M[N][N], в которой элемент M[i][j], содержит номер вершины k, полученной при нахождении наименьшего значения A[i][j] в k-ом проходе. Если A[i][j] =0, то вершины смежны.

Анализ. Так как алгоритм реализуется в три цикла по к, по I, по j, то временная сложность O(N3).

Пример



До первой рекурсии внешнего цикла, обозначенного выше k = 0, единственные известные пути соответствуют отдельным ребрам в графе.

При k = 1 находятся пути, проходящие через вершину 1: в частности, найден путь [2,1,3], заменяющий путь [2,3], который имеет меньше ребер, но длиннее (с точки зрения веса).

При k = 2 находятся пути, проходящие через вершины 1,2. Красные и синие прямоугольники показывают, как путь [4,2,1,3] собирается из двух известных путей [4,2] и [2,1,3], встреченных в предыдущих итерациях, с 2 на пересечении. Путь [4,2,3] не рассматривается, потому что [2,1,3] - это кратчайший путь, встреченный до сих пор от 2 до 3.

При k = 3 пути, проходящие через вершины 1,2,3 найдены.

Наконец, при k = 4 находятся все кратчайшие пути.

Матрица расстояний на каждой итерации k, обновленные расстояния выделены жирным шрифтом, будет иметь вид:

$k=0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=1$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=2$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k=3$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k=4$		j			
		1	**2**	**3**	**4**
i	**1**	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

Остовные деревья

Остовное дерево графа

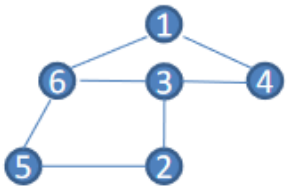
Определение 1. Деревом называется произвольный связный не ориентированный граф без циклов.

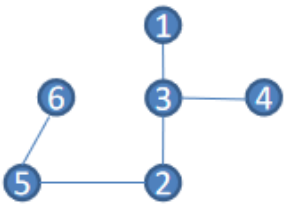
Определение 2. Связный граф, содержащий N вершин и N-1 ребер.

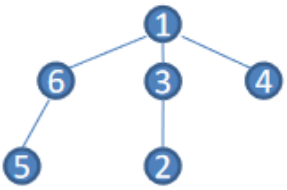
Определение 3. Для произвольного связного не ориентированного графа G{V,E} каждое дерево {V,T} где $T \subseteq E$ называется стягивающим деревом или каркасом или остовным деревом
Ребро такого дерева называется ветвями, а остальные ребра графа хордами.

Полный граф с N вершинами содержит N^{N-2} остовных деревьев. Поиск каждого занимает O(e) времени (e – количество ребер). В полном дереве e=N(N-1)/2.Тогда решение задачи прямым поиском имело бы время $O(N^N)$*

Методы поиска в глубину и ширину в графе позволяют построить одиночный каркас. В эти программы можно вставить запись данных по ребрам, связывающим текущую вершину с ранее не просмотренными в некую структуру данных – например в массив ребер Tree (int [2][N])







Метод Дж. Крускала
Построение минимального остовного дерева неориентированного связного графа

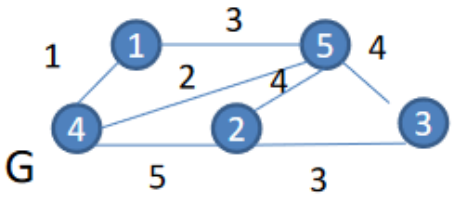
Дано: связный взвешенный неориентированный граф G{V,E}, ребра имеют вес.

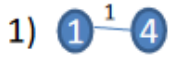
Результат: Остовное дерево с минимальным суммарным весом Q{V,T} где $\overline{T \subsetneq E}$

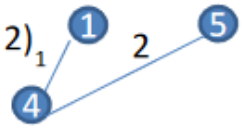
Подход к решению
Представление графа: Перечень ребер с указанием их веса
Данный метод строит ациклический граф – Остовное дерево

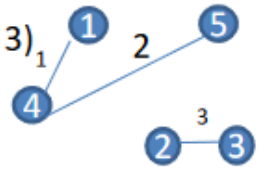
1. Граф Q изначально содержит только вершины, т.е. Q{V,0}
2. Ребра графа G упорядочены в порядке не убывания их весов.
3. Начинаем строить дерево Q с первого ребра графа G, добавляя далее ребра в граф Q, соблюдая условие: добавление ребра не должно приводить к появлению цикла в Q.
4. Повторение с п.3 до тех пор, пока число ребер в Q не станет равным V-1. Получившийся ациклический граф Q будет каркасом графа G минимального веса.

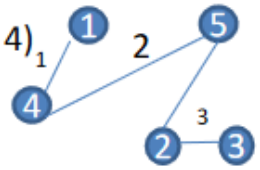
Рассмотрим алгоритм на примере











P – список ребер и их веса графа G – массив P[3][V*(V-1)]

1	**1**	**4**	**4**	**2**	**2**	**5**
5	4	5	2	5	3	3
3	1	2	5	4	3	4

Упорядоченный по весам список ребер

1	**4**	**1**	**2**	**2**	**5**	**4**
4	5	5	3	5	3	2
1	2	3	3	4	4	5

Итерация	Ребро	Mark
		{1,2,3,4,5}
1	<1,4>	{1,2,3,1,5}
2	<4,5>	{1,2,3,1,1}
3	<2,3>	{1,2,1,1,1}
4	<2,5>	{1,1,1,1,1}

Mark{1,2,3,4,5}- начальное значение массива меток вершин графа. Учитывает появление циклов и отвергает их: ребра из графа выбираются в каркас, если вершины, соединяемые им, имеют разные значения. Метка левой вершины должна быть меньше правой, если это не так, то ребро заменяется на параллельное (было 4-2, будет 2-4), так как граф не ориентированный это справедливо.

Ребро 1-4 первое ребро, отмечаем в Mark значением 1.
Ребро 4-5 вставляем в каркас и отмечаем в Mark значением 1.
Ребро 1-5 образует цикл, его в каркас не вставляем (отмечены желтым)

Алгоритм
M –количество ребер графа V –количество вершин
1. For i←1 to V do Mark[i]=i od
2. Просматриваем ребра из P и либо вставляем в граф Q либо нет
3. t ←M k ←0 –текущее ребро
4. While (k <V-1)
do
  i ←1
  ищем ребро у которого левая и праваяметки вершин разные (не создадут
  цикл)
  while(i<=t)and (Mark[P[1,i]]=Mark[P[2,i]]) and (P[1,i]≠0) do
    i++
  od
  запоминаем ребро каркаса
    L ←Mark[P[1,i]] R ←Mark[P[2,i]]
    строим если нужно параллельное ребро
    if (R<L) do
      temp ← L ; L ←R; R←temp
    od
    For j←1 to V do
      if Mark[i]=R Then Mark[i]=1
    od
od

Метод Прима

Построение минимального остовного дерева неориентированного связного графа

Постановка задачи
Дано: связный взвешенный неориентированный граф G{V,E}, ребра имеют вес.
Результат: Остовное дерево с минимальным суммарным весом Q{V,T} где $T \subsetneq E$

Подход к решению
Представление графа: Матрица смежности A[V][V] . Элемент матрицы со значением не равным 0 определяет вес ребра.
Данный метод строит на каждом шаге дерево (а не ациклический граф как метод Крускала) –т.е. добавляет ребро с минимальным весом , одна вершина которого уже принадлежит остовному дереву, а другая нет.

Такой принцип подключения исключает возможность появления циклов.

Для реализации метода используются два множества из 1..V элементов
SM –содержит список не выбранных в дерево вершин графа. Первоначально SM{1,2,..V}
SP –содержит список уже включенных в дерево вершин.

Если ребро i-j включается в дерево T, то один из номеров вершин i или j исключается из SM и включается в SP, кроме первого шага, когда оба номера заносятся в SP и исключаются из SM.

Это наиболее простой алгоритм построения минимального остовного дерева графа. Он похож на алгоритм Дейкстры поиска кратчайших путей.

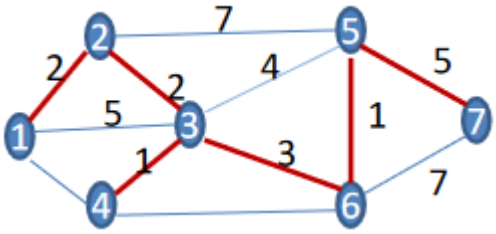
Алгоритм по Методу Прима

1. Выбирается некоторая вершина v остальные N-1 вершины отмечаются как не выбранные

2. Выбираем вершину с минимальным весом до нее от вершины v (т.е. ищем ребро с минимальным весом) и фиксируем выбранное ребро и его вес.

3. Выбранную вершину исключаем из списка выбранных включаем в список включенных в дерево.

4. Выполнение повторяется с пункта 1 до тех пор, пока не будут выбраны все вершины, т.е. (N-1) раз.



2	1	2	3	4	5	6	7
1	0	2	5		0	0	0
2	2	0	0	0	7	0	0
3	5	2	0	4	4	3	0
4		0	1	0	0	4	0
5	0	7	4	0	0	1	5
6	0	0	3	4	1	0	7
7	0	0	0	0	5	7	0

SM	1	2			5	6	7
SP	3	4					

SM	1				5	6	7
SP	3	4	2				

SM					5	6	7
SP	3	4	2	1			

SM					5		7
SP	3	4	2	1	6		

SM							7
SP	3	4	2	1	6	5	

SM							7
SP	3	4	2	1	6	5	

SM							
SP	3	4	2	1	6	5	7

Алгоритм
SM{1,2,3,4,5,6,7} SP{}
0. Первое ребро с минимальным весом – поиск первого минимального значения в матрице и сохранение вершин ребра (координат элемента массива) и веса.

Получаем ребро 3-4. SM{1,2,5..6} SP(3,4}

Строим дерево
1. Пока множество SM не станет пустым, т.е. все узлы не будут выбраны, повторяем:
2. Ищем ребро с минимальным весом среди элементов матрицы, с координатами i,j где i не включенная в SP вершина, а j включенная и A[i][j]≠0. Запоминаем координаты ребра и вес: L=I и t=j
min=A[i,j];
3. Вставляем L в SP и исключаем L из SM. Можем вывести ребро <L,t>
4. Повторяем с п.1.

Цикл в графе

Гамильтоновым путём называется простой путь, проходящий через каждую вершину графа ровно один раз.

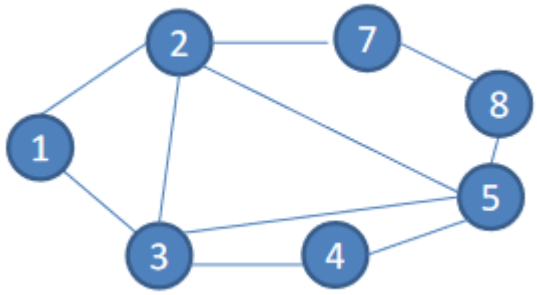
Гамильтоновым циклом называют замкнутый гамильтонов путь. Поиск гамильтоновых циклов – перебор с возвратом.

Граф называется гамильтоновым, если он содержит гамильтонов цикл.

Эйлеров цикл – это такой цикл, который проходит ровно один раз по каждому ребру.

Эйлеров цикл в графе существует тогда и только тогда, когда граф связный и все его вершины имеют четные степени.

Граф, содержащий эйлеровый цикл, называется **эйлеровым графом**



Поиск эйлерова цикла на матрице смежности

Используется поиск в глубину.
При этом ребра удаляются.
Порядок просмотра (номера вершин) запоминаются.
При обнаружении вершины, из которой не выходят ребра – она удаляется, ее номер записывается в стек и просмотр продолжается от предыдущей вершины.
Обнаружение вершины с нулевым числом ребер говорит о том, что цикл найден. Его можно удалить, четность вершин при этом не изменится.
Процесс продолжается, пока есть ребра. В стеке запоминаются вершины в порядке соответствующем циклу

Гамильтоновы циклы