数据结构-Dijistra

信智学院　陈琰宏

主要内容

01 Dijkstra算法

02 优先队列

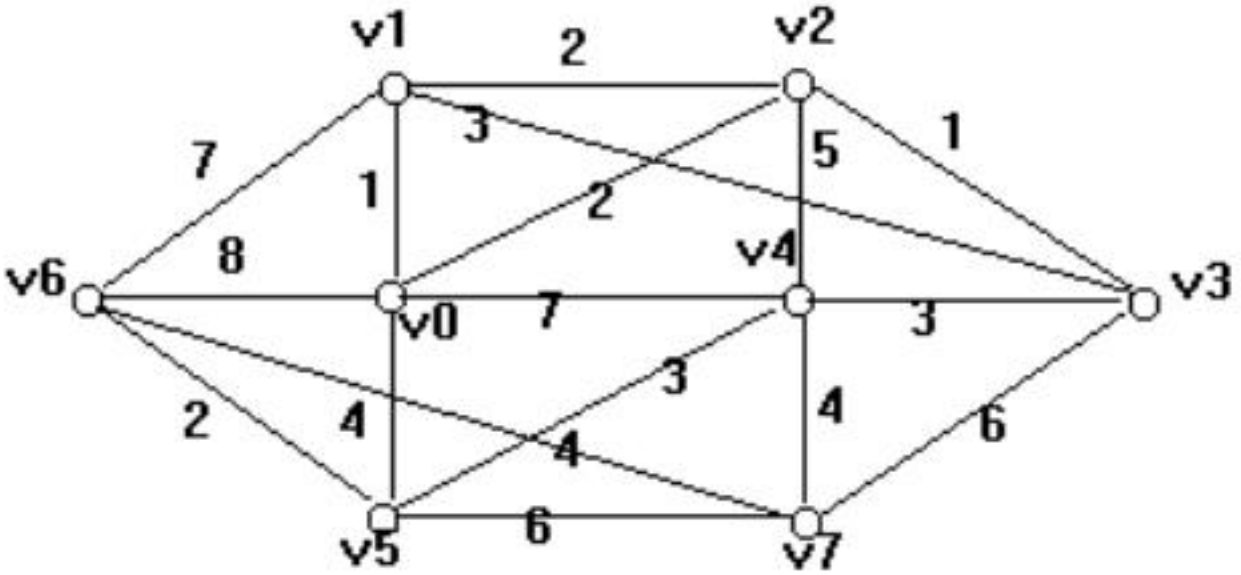
03 案例实现

1 最短路径

交通的**最短路径**选择。

1 最短路径

8个城市 v_0，v_1，…，v_7之间有一个公路网(如图所示)，每条公路为图中的边，边上的权数表示通过该公路所需的时间．设你处在城市 v_0，那么从 v_0到其他各城市，应选择什么路径使所需的时间最短？



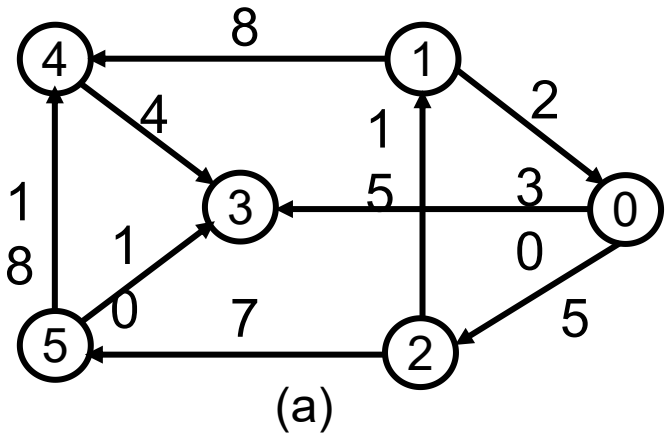
1 最短路径

- 最短路径问题：如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径，使得沿此路径各边上的权值总和达到最小。
- 问题解法：
  1. 权值为非负的单源最短路径问题(固定源点)—Dijkstra算法(迪杰斯特拉算法)；
  2. 权值为任意值的单源最短路径问题(固定源点) —Bellman-Ford算法(贝尔曼—福特算法)；
  3. 所有顶点之间的最短路径问题—Floyd-Warshall算法(弗洛伊德算法)；

1.1 Dijkstra算法

权值为非负的单源最短路径问题

- 问题的提出： 给定一个带权有向图G与源点v，求从v到G中其它顶点的最短路径。限定各边上的权值大于或等于0。



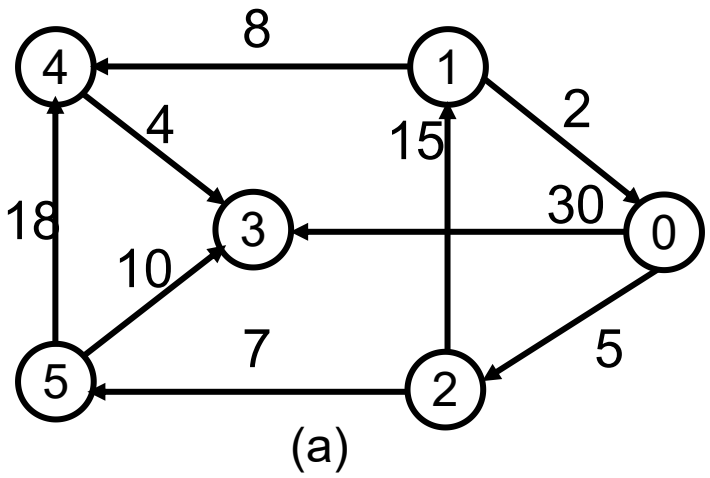
在图(a)中，考虑顶点0到其他顶点的最短距离是多少？

顶点0到顶点1的最短路径距离是：20
顶点0到顶点2的最短路径距离是： 5
顶点0到顶点3的最短路径距离是： 22
顶点0到顶点4的最短路径距离是： 28
顶点0到顶点5的最短路径距离是： 12

思考：这些最短距离是怎么求出来的？

1.1 Dijkstra算法

- 为求得这些最短路径，Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点v到其它各顶点的最短路径全部求出为止。



(a)

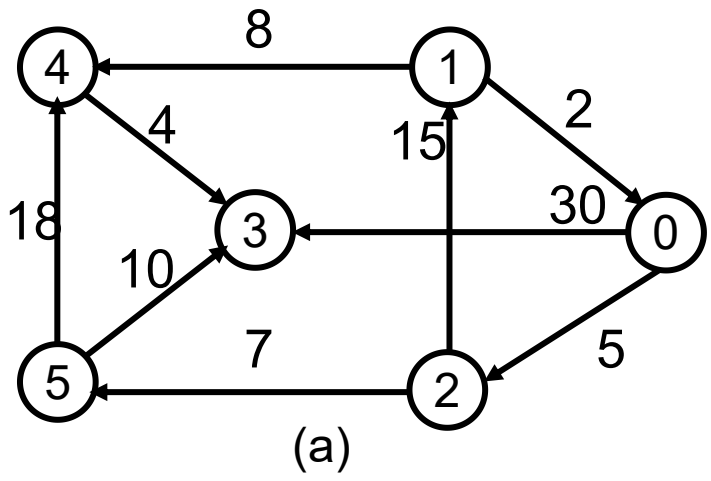
在图(a)中，考虑顶点0到其他顶点的最短距离是多少？

长度最短的最短路径是顶点0到顶点2的最短路径(就是顶点0到其他顶点的直接路径最短的路径)

长度次短的最短路径是顶点0到顶点5的最短路径(v_0,v_2,v_5)，其中(v_0,v_2)就是前面求出的长度最短的最短路径。

1.1 Dijkstra算法

- 为求得这些最短路径，Dijkstra提出**按路径长度的递增次序，逐步产生最短路径**的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点v到其它各顶点的最短路径全部求出为止。

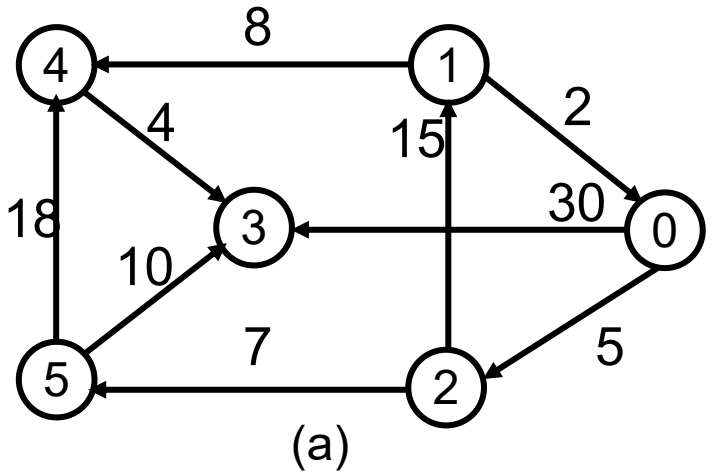


(a)

$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

1.1 Dijkstra算法的基本思想

1. 设置两个顶点的集合T和S：
   a. S中存放已找到最短路径的顶点，初始状态时，集合S中只有一个顶点，即源点v_0；
   b. T中存放当前还未找到最短路径的顶点；
2. 以后每求得一条最短路径$(v_0,\ldots,v_k)$，就将v_k加入到顶点集合S中，直到所有的顶点都加入到集合S中，算法就结束了。

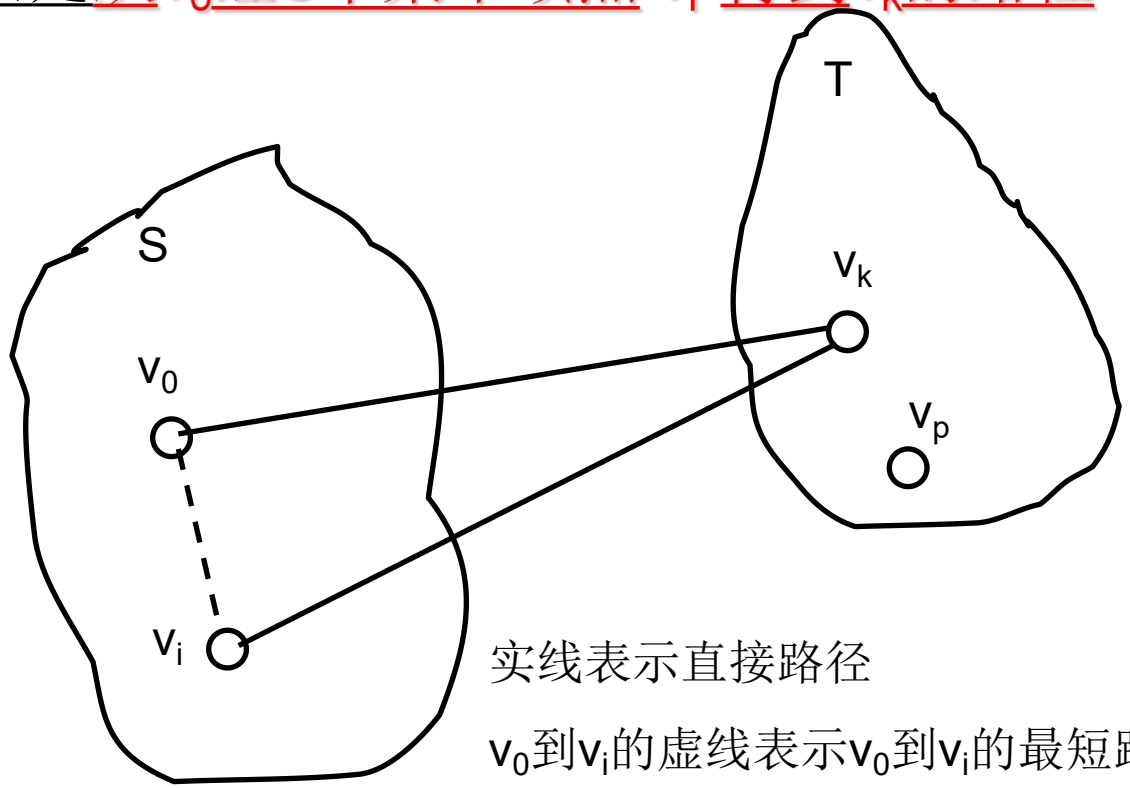


(a)

$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

1.1 Dijkstra算法的基本思想

- 可以证明v_0到T中顶点v_k的最短路径，要么是从v_0到v_k的直接路径；要么是从v_0经S中某个顶点 v_i 再到v_k的路径。



实线表示直接路径

v_0到v_i的虚线表示v_0到v_i的最短路径

算法的具体步骤

1. 初始时，S中包含源点v_0。
2. 然后不断从T中选取到顶点v_0路径长度最短的顶点u，找到后：
   a) **把顶点u加入到S；**
   b) **修改顶点v_0到T中剩余顶点的最短路径长度值。T中各顶点新的最短路径长度值为：min{原来的最短路径长度值，顶点u的最短路径长度+u到该顶点的路径长度值}；**
3. 重复2，直到T的顶点全部加入S为止。

算法的具体步骤

在dijkstra算法里设置2个数组：

1) dist[n]：dist[i]表示当前找到的从源点v_0到终点v_i的最短路径的长度，初始状态下，dist[i]为$E[v_0][i]$，即邻接矩阵的第v_0行。
2) S[n]：S[i]为0表示顶点v_i还未加入到集合S中，S[i]为1表示v_i已经加入到集合S中。初始状态下，$S[v_0]$为1，其余为0，表示最初集合S中只有顶点v_0。

算法的具体步骤

在dijkstra算法里重复做以下工作：

1) 在dist[ ]里查找S[i]!=1，并且dist[i]最小的顶点u，
2) 将S[u]改为1，表示顶点u已经加入进来了。
3) 修改其它顶点的dist[ ]。当S[k]!=1，且顶点v_k到顶点v_u有边(E[u][k]<MAX)，且dist[u]+E[u][k]<dist[k]，则修改dist[k]为dist[u]+E[u][k]。

递推公式：

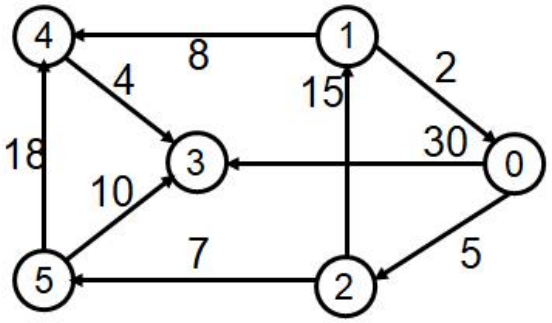
初始：*dist* [k] = Edge[v_0][k]，v_0是源点

递推：*dist* [k] = min{ *dist* [k], *dist* [u] + Edge[u][k] }, $v_k \in T$

其中v_u是当前dist[ ]最小的顶点。

算法的具体步骤

源点为0



$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

1. 在T选取dist[ ]最小的顶点u

2. 将u加入到S

3. 修改T中顶点的dist[ ]

初始状态

	S	dist
0	1	0
1	0	∞
2	0	5
3	0	30
4	0	∞
5	0	∞

(1)

	S	dist
0	1	0
1	0	20
2	1	5
3	0	30
4	0	∞
5	0	12

(2)

	S	dist
0	1	0
1	0	20
2	1	5
3	0	22
4	0	30
5	1	12

(3)

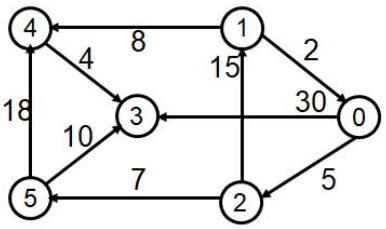
	S	dist
0	1	0
1	1	20
2	1	5
3	0	22
4	0	28
5	1	12

(4)

	S	dist
0	1	0
1	1	20
2	1	5
3	1	22
4	1	28
5	1	12

(5)

算法的具体步骤



$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

初始状态

	S	dist	path
0	1	0	-1
1	0	∞	-1
2	0	5	0
3	0	30	0
4	0	∞	-1
5	0	∞	-1

(1)

	S	dist	path
0	1	0	-1
1	0	**20**	**2**
2	1	***5***	0
3	0	30	0
4	0	∞	-1
5	0	**12**	**2**

(2)

	S	dist	path
0	1	0	-1
1	0	20	2
2	1	5	0
3	0	**22**	**5**
4	0	**30**	**5**
5	1	***12***	2

(3)

	S	dist	path
0	1	0	-1
1	1	***20***	2
2	1	5	0
3	0	22	5
4	0	**28**	**1**
5	1	12	2

(4)

	S	dist	path
0	1	0	-1
1	1	20	2
2	1	5	0
3	1	***22***	5
4	0	28	1
5	1	12	2

(5)

	S	dist	path
0	1	0	-1
1	1	20	2
2	1	5	0
3	1	22	5
4	1	***28***	1
5	1	12	2

算法的具体代码

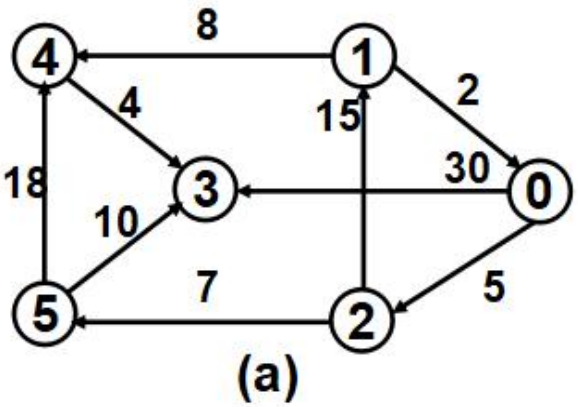
该算法可以实现的功能为：指定一个源点，指定一个终点，求源点到终点的最短路径。

```
#define max_v_num 10
#define max 1000000
int Edge[max_v_num][max_v_num];
int vexnum;
/*
dist[i]表示当前找到的从源点v0到终点vi的最短路径的长度，
初始状态下，dist[i]为E[v0][i]，即邻接矩阵的第v0行。
S[i]为0表示顶点vi还未加入到集合S中，S[i]为1表示vi已经加入到集合S中。
初始状态下，S[v0]为1，其余为0，表示最初集合S中只有顶点v0。
*/
```

算法的具体代码

```
13 void Dijistra(int v, int end){//v 源点 end终点
14     int dis[max_v_num],S[max_v_num];
15     int i,j,k;
16     for(i=0;i<vexnum;i++){
17         dis[i]=Edge[v][i];//初始化dist[i]
18         S[i]=0;//让所有的顶点归到一个初始集合T
19     }
20     S[v]=1,dis[v]=0;//把源点v加入到集合S中，初始化路径数组
21
22     for(i=0;i<vexnum-1;i++){//寻路除源点外的其他顶点
23         int min=max,u=v;
24         for(j=0;j<vexnum;j++){
25             if(S[j]==0&&dis[j]<min){u=j;min=dis[j];}
26         }//1 找到当前的最短路径
27
28         S[u]=1;//2 把找到的顶点加入到集合S中
29
30         for(k=0;k<vexnum;k++){//u点加入到集合T后,多出了新的路径,
31             //即通过u点可以达到新的顶点，更新dis[]数组
32             if(S[k]==0&&Edge[u][k]<max&&dis[u]+Edge[u][k]<dis[k]){
33                 dis[k]=dis[u]+Edge[u][k];
34             }
35         }
36     }
37     cout<<dis[end];
38 }
```

1 [3333]: 迪杰斯特拉



$$\begin{bmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{bmatrix}$$

输入顶点数n 边数m，m条边的顶点和权值；输出：顶点0到每个顶点的最短路径

样例输入

```
6 9
0 2 5
0 3 30
1 0 2
1 4 8
2 1 15
2 5 7
4 3 4
5 3 10
5 4 18
0 4
```

样例输出

```
28
```

```
34 int main(){
35     int i,j;
36     int n,m;//顶点数和边数
37     int u,v,w;//边的起始点和改边的权值
38     int x,y;//源点和终点
39     cin>>n>>m;
40     vexnum=n;
41     for(i=0;i<n;i++)
42         for(j=0;j<n;j++)
43             if(i==j)Edge[i][j]=0;
44             else Edge[i][j]=max;
45     for(i=0;i<m;i++){
46         cin>>u>>v>>w;
47         Edge[u][v]=w;
48     }
49     cin>>x>>y;
50     Dijistra(x,y);
51     return 0;
52 }
```

```
1  #include<iostream>
2  using namespace std;
3  #define max_v_num 10
4  #define max 1000000
5  int Edge[max_v_num][max_v_num];
6  int vexnum;
7  /*
8  dist[i]表示当前找到的从源点v0到终点vi的最短路径的长度，
9  初始状态下，dist[i]为E[v0][i]，即邻接矩阵的第v0行。
10 S[i]为0表示顶点vi还未加入到集合S中，S[i]为1表示vi已经加入到集合S中。
11 初始状态下，S[v0]为1，其余为0，表示最初集合S中只有顶点v0。
12 */
```

```
13 void Dijistra(int v, int end){//v 源点 end终点
14     int dis[max_v_num],S[max_v_num];
15     int i,j,k;
16     for(i=0;i<vexnum;i++){
17         dis[i]=Edge[v][i];//初始化dist[i]
18         S[i]=0;//让所有的顶点归到一个初始集合T
19     }
20     S[v]=1,dis[v]=0;//把源点v加入到集合S中，初始化路径数组
21
22     for(i=0;i<vexnum-1;i++){//寻路除源点外的其他顶点
23         int min=max,u=v;
24         for(j=0;j<vexnum;j++){
25             if(S[j]==0&&dis[j]<min){u=j;min=dis[j];}
26         }//1 找到当前的最短路径
27
28         S[u]=1;//2 把找到的顶点加入到集合S中
29
30         for(k=0;k<vexnum;k++){//u点加入到集合T后,多出了新的路径,
31             //即通过u点可以达到新的顶点，更新dis[]数组
32             if(S[k]==0&&Edge[u][k]<max&&dis[u]+Edge[u][k]<dis[k]){
33                 dis[k]=dis[u]+Edge[u][k];
34             }
35         }
36     }
37     cout<<dis[end];
38 }
```

比较： Prim和Dijkstra的区别

Prim和Dijkstra大同小异，唯一的区别是prim是维护树到其余节点的最小路径，而dijkstra是维护源节点到各个节点的最小路径。

比较：最小生成树和最短路径的区别

最小生成树：

最小生成树能够保证整个拓扑图的所有路径之和最小，但不能保证任意两点之间是最短路径。

最短路径：

最短路径是从一点出发，到达目的地的路径最小。

总结：

- 遇到求所有路径之和最小的问题用最小生成树&并查集解决；
- 遇到求两点间最短路径问题的用最短路，即从一个城市到另一个城市最短的路径问题。

区别：

最小生成树构成后所有的点都被连通，而最短路只要到达目的地走的是最短的路径即可，与所有的点连不连通没有关系。

算法复杂度分析

用邻接矩阵存储图，Dijkstra算法的复杂度为$O(n^2)$

能不能降低时间复杂度？

采用邻接表+优先队列

独立思考与判断

2 优先队列

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。

在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （first in, largest out）的行为特征。

2.1 优先队列用法

1.头文件

#include<queue>//与队列相同，不必引入vector的头文件

2.定义方式

priority_queue<int> p;//默认，最大值优先，是大顶堆一种简写方式
priority_queue<int,vector<int>,greater<int>>q1;//最小值优先，小顶堆
priority_queue<int,vector<int>,less<int> >q2;//最大值优先，大顶堆

```
//对于基础类型 默认是大顶堆
priority_queue<int> a;
//等同于 priority_queue<int, vector<int>, less<int> > a;

// 这里一定要有空格，不然成了右移运算符↓ ↓
priority_queue<int, vector<int>, greater<int> > c;  //这样就是小顶堆
priority_queue<string> b;
```

其中第一个参数是数据类型，第二个参数为容器类型。第三个参数为比较函数。

```cpp
#include<iostream>
#include <queue>
using namespace std;
int main()
{
    //对于基础类型 默认是大顶堆
    priority_queue<int> a;
    //等同于 priority_queue<int, vector<int>, less<int> > a;

    // 这里一定要有空格，不然成了右移运算符↓ ↓
    priority_queue<int, vector<int>, greater<int> > c;  //这样就是小顶堆
    priority_queue<string> b;

    for (int i = 0; i < 5; i++){
        a.push(i);
        c.push(i);
    }
    while (!a.empty())
    {
        cout << a.top() << ' ';
        a.pop();
    }
    cout << endl;

    while (!c.empty())
    {
        cout << c.top() << ' ';
        c.pop();
    }
    cout << endl;

    b.push("abc");
    b.push("abcd");
    b.push("cbd");
    while (!b.empty())
    {
        cout << b.top() << ' ';
        b.pop();
    }
    cout << endl;
    return 0;
}
```

```
4 3 2 1 0
0 1 2 3 4
cbd abcd abc
```

2.1 优先队列用法

```
struct node
{
        int dis, num;
};
priority_queue<node>que;
bool operator<(const node &a, const node &b)
{
        return a.dis > b.dis;
}//按照dis从小到大排序
```

2.1 优先队列用法

```
struct node
{
    string name;
    int price;
    friend bool operator< (node a, node b)
    {
        return a.price < b.price;
      // 相当于less,这是大顶堆，反之则是小顶堆，最大值优先
    }
} stu; //定义结构体变量
```

这样直接可以：

```
priority_queue<node > q;
```

2.1 优先队列用法

q.push(x) //将x加入队列中，即入队操作

q.pop() //出队操作(删除队列首元素)，只是出队，没有返回值

q.top() //返回第一个元素(队首元素)优先队列的队首用top

q.size() //返回栈队列中的元素个数

q.empty() //当队列为空时，返回 true

2.3 Vector介绍

在使用数组的时候，必须指定数组的长度，一旦配置了就不能改变了，通常我们的做法是尽量配置一个大的空间，以免不够用，这样做的缺点是比较浪费空间，预估空间不当会引起很多不便。

STL实现了一个Vector容器，该容器就是来改善数组的缺点。vector是一个动态空间，随着元素的加入，它的内部机制会自行扩充以容纳新元素。因此，vector的运用对于内存的合理利用与运用的灵活性有很大的帮助，再也不必因为害怕空间不足而一开始就配置一个大容量数组了，vector是用多少就分配多少。

2.3 Vector介绍

使用vector容器之前必须加上<vector>头文件：#include<vector>;
vector成员函数

c.push_back(elem)在尾部插入一个elem数据。

```
vector<int> v;
v.push_back(1);
```

c.pop_back()删除末尾的数据。

```
vector<int> v;
v.pop_back();
```

c.assign(beg,end)将[beg,end)一个左闭右开区间的数据赋值给c。

2.3 Vector介绍

clear() 清空当前的vector
max_size() 得到vector最大可以是多大
empty() 判断vector是否为空
size() 当前使用数据的大小

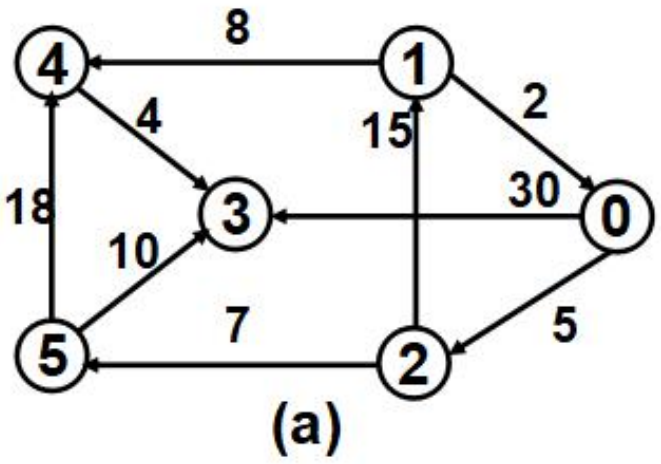
at 得到编号位置的数据
begin 得到数组头的指针
end 得到数组的最后一个单元+1的指针
front 得到数组头的引用
back 得到数组的最后一个单元的引用

capacity 当前vector分配的大小
resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值
reserve 改变当前vecotr所分配空间的大小
erase 删除指针指向的数据项

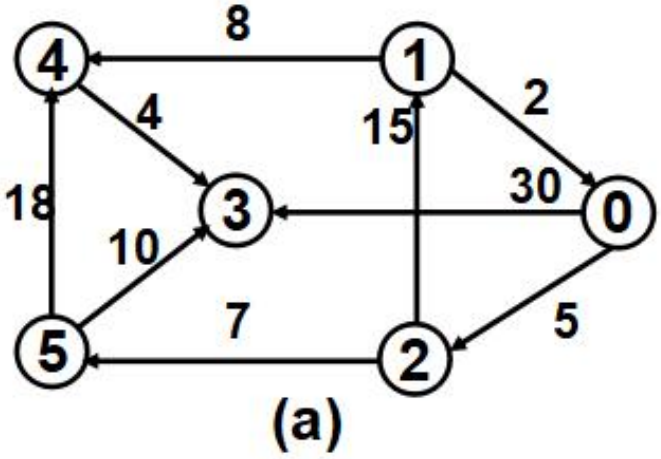
2.4 [3333]: 迪杰斯特拉

如图，求最短路径。

输入
顶点数n 边数m
m条边的顶点和权值
某两个顶点
输出
顶点0到每个顶点的最短路径
样例输入
6 9
0 2 5
0 3 30
1 0 2
1 4 8
2 1 15
2 5 7
4 3 4
5 3 10
5 4 18
0 4
样例输出
28



2.4.1 建立邻接表

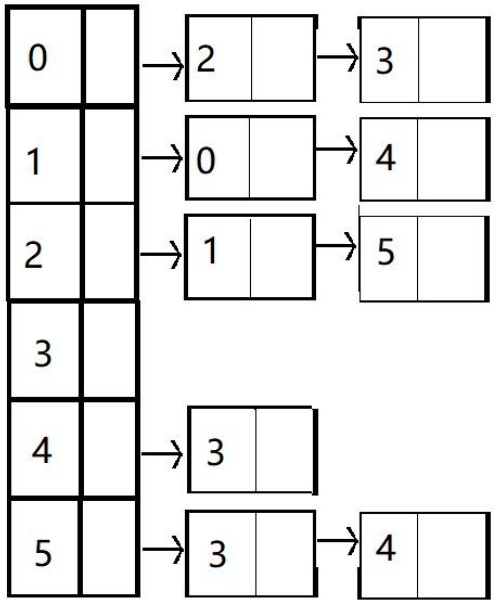


disc[MAXN]

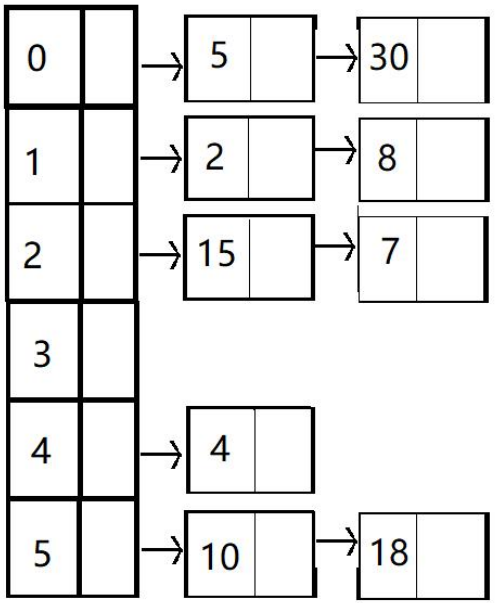
0x3f

0	0	0x3f	0x3f	0x3f	0x3f	0x3f
	0	1	2	3	4	5

g[MAXN] g[u].push_back(v)



c[MAXN] c[u].push_back(w)

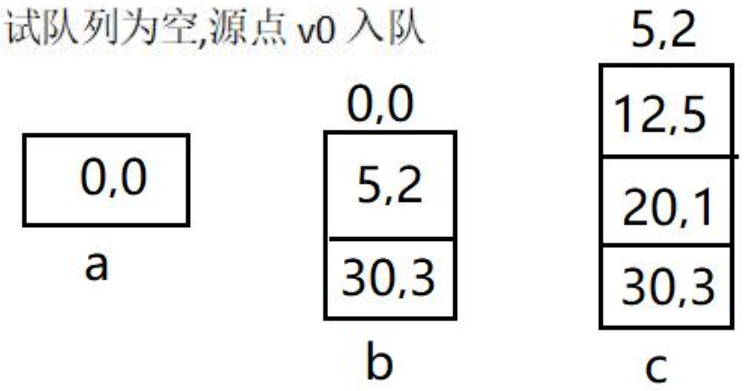


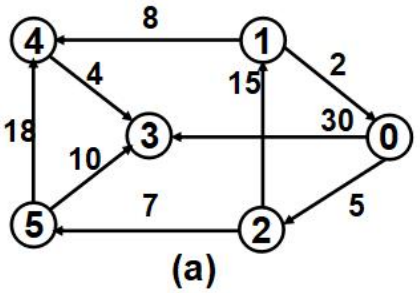
2.4.2 建立优先队列

```cpp
#include<bits/stdc++.h>
using namespace std;
const int inf=0xffffff;
const int MAXN=100;
int edge[MAXN][MAXN];//边信息
int disc[MAXN];//保存源点到各点的最短距离
vector<int>g[MAXN], c[MAXN];
struct node
{
    int dis;//dis表示源点到当前点num的距离
    int num; //编号
};
priority_queue<node>que;//优先队列
bool operator<(const node &a, const node &b)
{
    return a.dis > b.dis;//按照dis从小到大排
}
```

2.4.2 建立优先队列

1. 初试队列为空,源点 v0 入队





2. 从 V0 出发，走 V2 、V5。将 V2,V5 入队，根据优先队列原则，dis 最小的处于队头，同时更新 disc[]数组。

disc[MAXN]

	0	1	2	3	4	5
0	0	0x3f	5	30	0x3f	0x3f

3. 弹出 v2，从 V2 出发，走 V1,V5,更新优先队列如 c 所示，同时更新 disc[]数组

disc[MAXN]

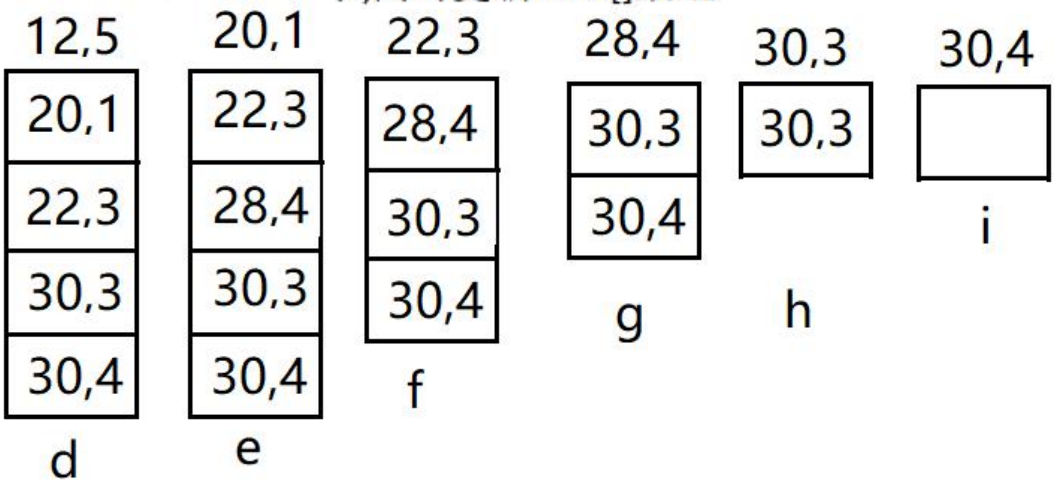
	0	1	2	3	4	5
0	0	20	5	30	0x3f	12

2.4.2 建立优先队列

4. 弹出 v5，从 V5 出发，走 V3, V4，更新优先队列如 d 所示,同时更新 disc[]数组

disc[MAXN]

	0	1	2	3	4	5
0	0	20	5	30	30	12

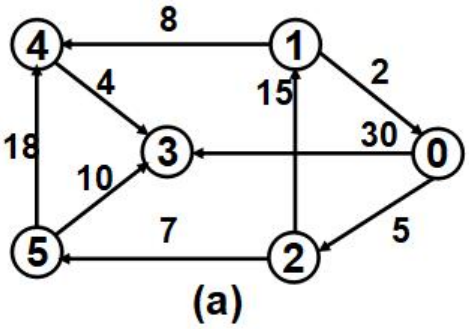


5. 弹出 v1, 从 v1 出发，走 V0,V4, 更新优先队列如 e 所示,同时更新 disc[]数组

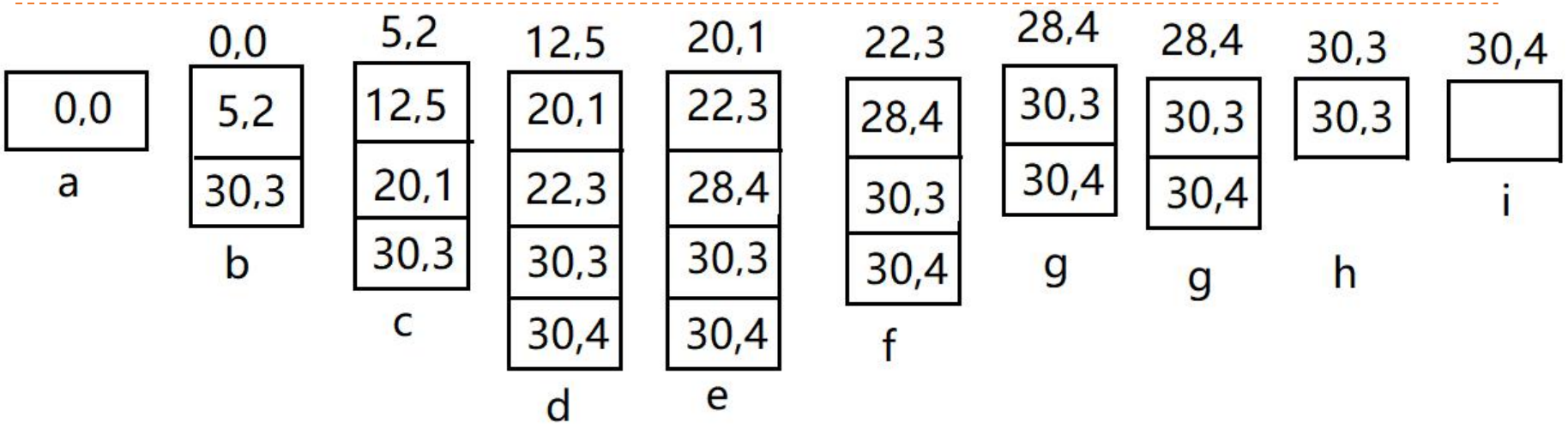
disc[MAXN]

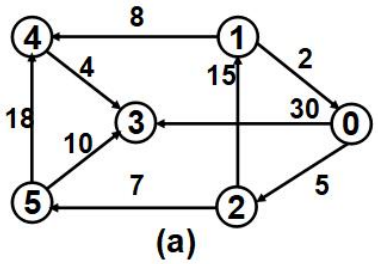
	0	1	2	3	4	5
0	0	20	5	30	28	12

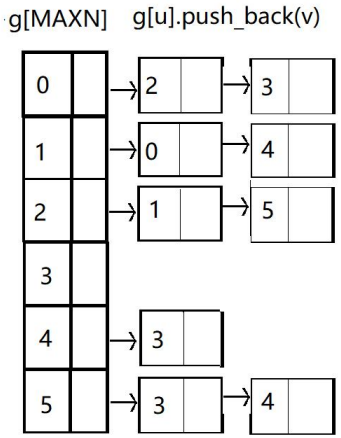
6. 弹出 V3,从 V3 出发，无路可走，更新优先队列如 f 所示。
7. 弹出 v4,从 v4 出发，disc[]没有更新,如图 g。
8. 到结束。

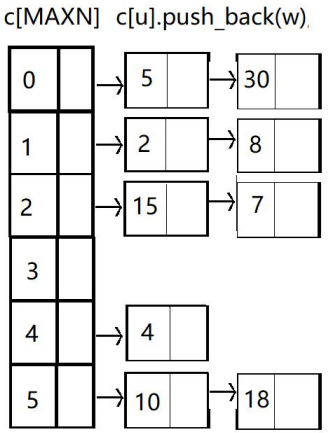


2.4.2 建立优先队列


0,0
a
0,0
5,2
30,3
b
5,2
12,5
20,1
30,3
c
12,5
20,1
22,3
30,3
30,4
d
20,1
22,3
28,4
30,3
30,4
e
22,3
28,4
30,3
30,4
f
28,4
30,3
30,4
g
28,4
30,3
30,4
g
30,3
30,3
h
30,4
i


4
1
3
0
5
2
8
4
15
2
18
30
10
7
5
(a)


g[MAXN]
g[u].push_back(v)
0
2
3
1
0
4
2
1
5
3
4
3
5
3
4


c[MAXN]
c[u].push_back(w)
0
5
30
1
2
8
2
15
7
3
4
4
5
10
18

2.4.3 代码

```
int main(){
    int n,m;
    int u,v,w;
    int st,ed;
    cin>>n>>m;
    //队尾添加元素：  vec.push_back();
    //队尾删除元素：  vec.pop_back();
    for(int i=0;i<m;i++){
        cin>>u>>v>>w;
        g[u].push_back(v);//u是邻接表出来的端点，uv是一条边
        c[u].push_back(w);//存权值
    }
    cin>>ed;
    Dijistra(0,ed);
    return 0;
}
```

2.4.3 代码

```
19  void Dijistra(int v, int end){
20      memset(disc, 0x3f, sizeof(disc));
21      disc[v] = 0;//初始化，源点到其他点初始距离为极大值
22      node tmp;
23      tmp.num = v; tmp.dis = 0;//初试源点
24      que.push(tmp);//源点入队
25      while(!que.empty())
26      {
27
28          node now = que.top();//得到dis最小点
29          que.pop();
30          if(now.dis != disc[now.num])
31              continue;//disc数组中存在的最短距离，如果取出来的边不是最小的就
32          for(int i = g[now.num].size() - 1; i >= 0; i --)
33          {//遍历邻接表
34              int to = g[now.num][i];
35              if(disc[to] > disc[now.num] + c[now.num][i])
36              {
37                  disc[to] = disc[now.num] + c[now.num][i];
38                  que.push(node{disc[to], to});
39              }
40          }
41      }
42      cout << disc[end] << endl;
43  }
```

2.4.3 代码

```
19 void Dijistra(int v, int end){
20     memset(disc, 0x3f, sizeof(disc));
21     disc[v] = 0;//初始化，源点到其他点初始距离为极大值
22     node tmp;
23     tmp.num = v; tmp.dis = 0;//初始源点
24     ____________________//源点入队
25     while(!que.empty())
26     {
27
28         node now = ________________;//得到dis最小点
29         que.pop();
30         if(now.dis != disc[now.num])
31             continue;//disc数组中存在的最短距离，如果取出来的边不是最小的就
32         for(_____________________________)
33         {//遍历邻接表
34             int to = g[now.num][i];
35             if(_____________________________)
36             {
37                 disc[to] = _____________________________;
38                 que.push(node{disc[to], to});
39             }
40         }
41     }
42     cout << disc[end] << endl;
43 }
```

3 SPFA算法

SPFA算法是求单源最短路径的一种算法，它是Bellman-ford的队列优化，它是一种十分高效的最短路算法。很多时候，给定的图存在负权边，这时类似Dijkstra等算法便没有了用武之地，而Bellman-Ford算法的复杂度又过高，这时SPFA算法便派上用场了。
SPFA的复杂度大约是O(kE),k是每个点的平均进队次数(一般的，k是一个常数，在稀疏图中小于2)。
但是，SPFA算法稳定性较差，在稠密图中SPFA算法时间复杂度会退化。

SPFA算法

实现方法：

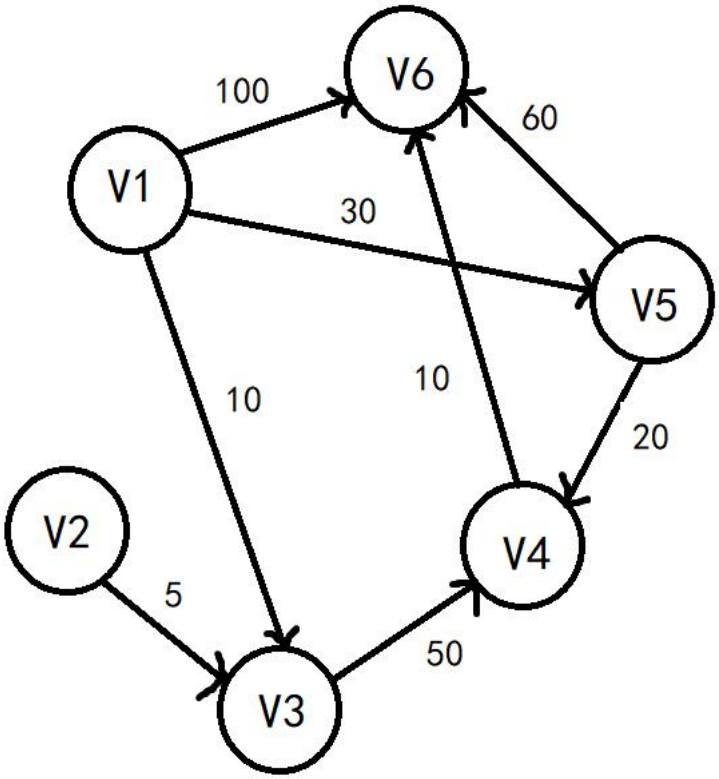
1：建立一个队列，初始时队列里只有起始点，

2：再建立一个表格记录起始点到所有点的最短路径（该表格的初始值要赋为极大值，该点到他本身的路径赋为0）。

3：然后执行松弛操作，用队列里有的点去刷新起始点到所有点的最短路，如果刷新成功且被刷新点不在队列中则把该点加入到队列最后。

4：重复执行3直到队列为空。

SPFA算法模拟

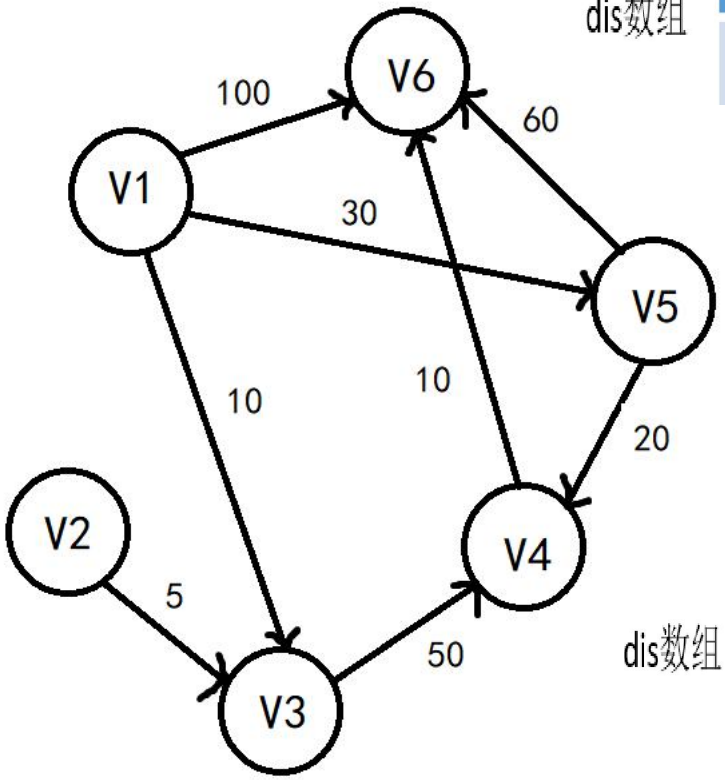


首先我们先初始化数组dis如下图所示：（除了起点赋值为0外，其他顶点的对应的dis的值都赋予无穷大，这样有利于后续的松弛）

dis数组

v1	v2	v3	v4	v5	v6
0	∞	∞	∞	∞	∞

此时，我们还要把v1入队列：{v1}
现在进入循环，直到队列为空才退出循环。



dis数组

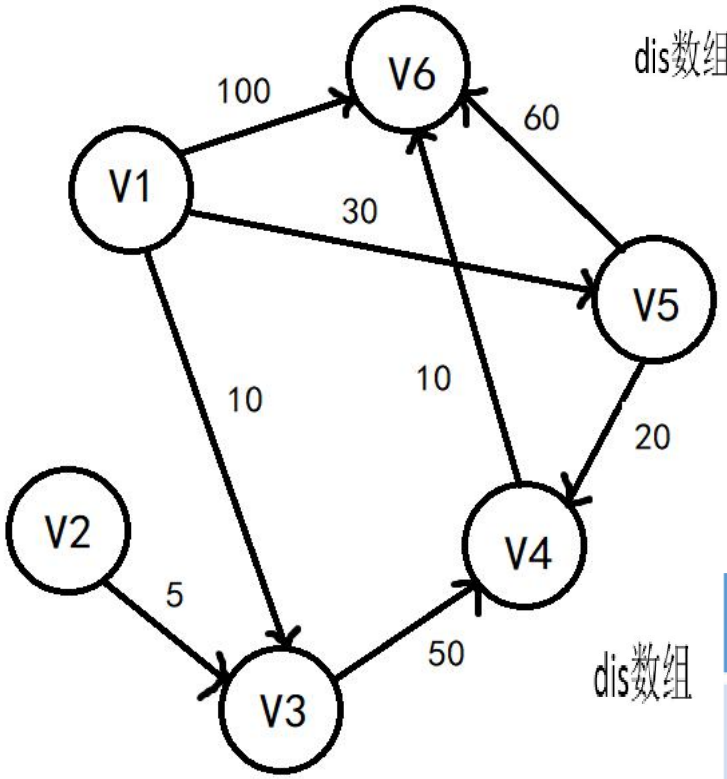
v1	v2	v3	v4	v5	v6
0	∞	∞	∞	∞	∞

第一次循环：
首先，队首元素出队列，即是v1出队列，然后，对以v1为弧尾的边对应的弧头顶点进行松弛操作，可以发现v1到v3，v5，v6三个顶点的最短路径变短了，更新dis数组的值，得到如下结果：

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	∞	30	100

我们发现v3，v5，v6都被松弛了，而且不在队列中，所以要它们都加入到队列中：{v3，v5，v6}



dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	∞	30	100

第二次循环

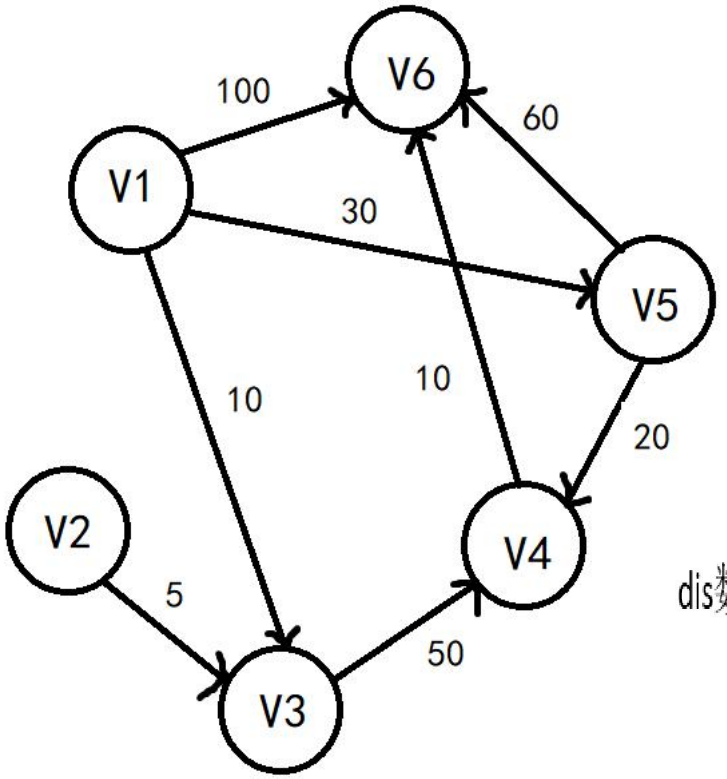
此时，队首元素为v3，v3出队列，然后，对以v3为弧尾的边对应的弧头顶点进行松弛操作，可以发现v1到v4的边，经过v3松弛变短了，所以更新dis数组，得到如下结果：

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	60	30	100

此时只有v4对应的值被更新了，而且v4不在队列中，则把它加入到队列中：

{v5, v6, v4}



dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	60	30	100

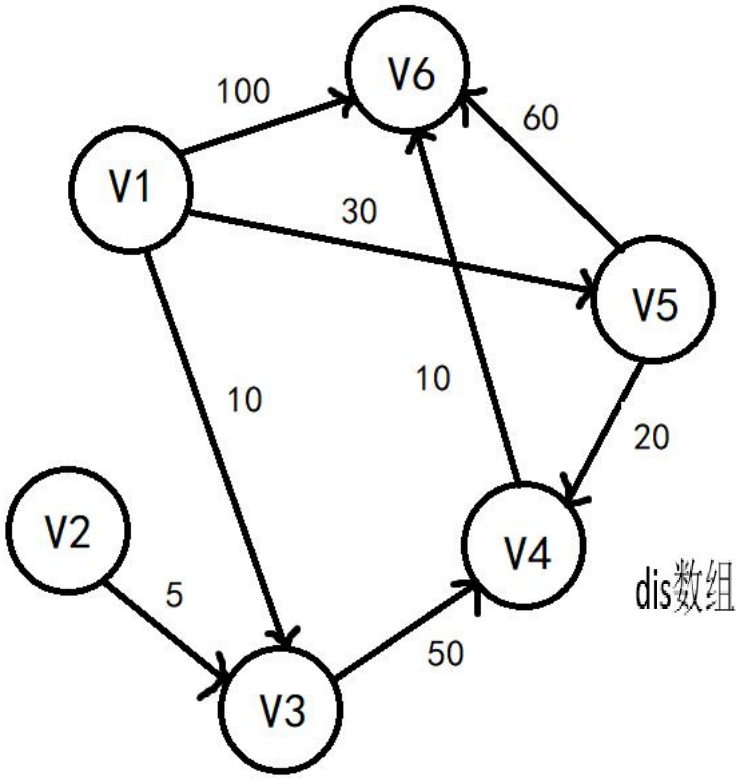
第三次循环

此时，队首元素为v5，v5出队列，然后，对以v5为弧尾的边对应的弧头顶点进行松弛操作，发现v1到v4和v6的最短路径，经过v5的松弛都变短了，更新dis的数组，得到如下结果：

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	90

我们发现v4、v6对应的值都被更新了，但是他们都在队列中了，所以不用对队列做任何操作。队列值为：{v6，v4}



dis数组

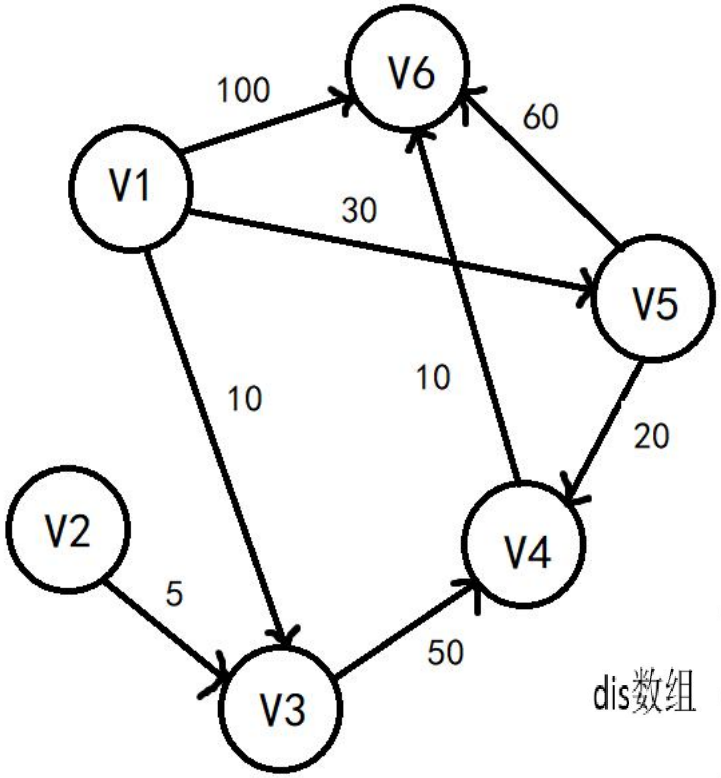
第四次循环

此时，队首元素为v6，v6出队列，然后，对以v6为弧尾的边对应的弧头顶点进行松弛操作，发现v6出度为0，所以我们不用对dis数组做任何操作，其结果和上图一样，队列同样不用做任何操作，它的值为：{v4}

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	90

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	90



第五次循环

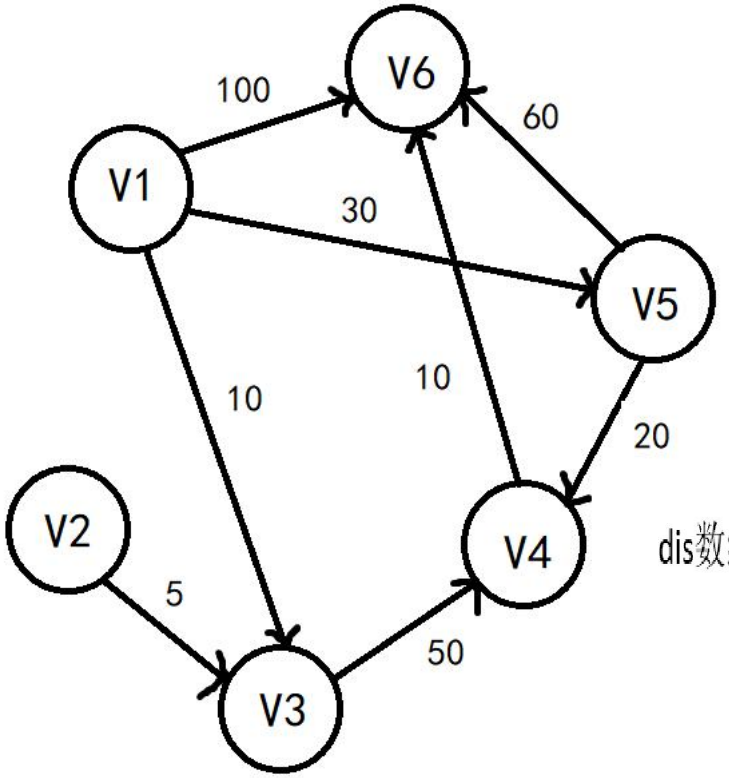
此时，队首元素为v4，v4出队列，然后，对以v4为弧尾的边对应的弧头顶点进行松弛操作，可以发现v1到v6的最短路径，经过v4松弛变短了，所以更新dis数组，得到如下结果：

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	60

dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	60



第六次循环

此时，队首元素为v6，v6出队列，然后，对以v6为弧尾的边对应的弧头顶点进行松弛操作，发现v6出度为0，所以我们不用对dis数组做任何操作，其结果和上图一样，队列同样不用做任何操作。所以此时队列为空。

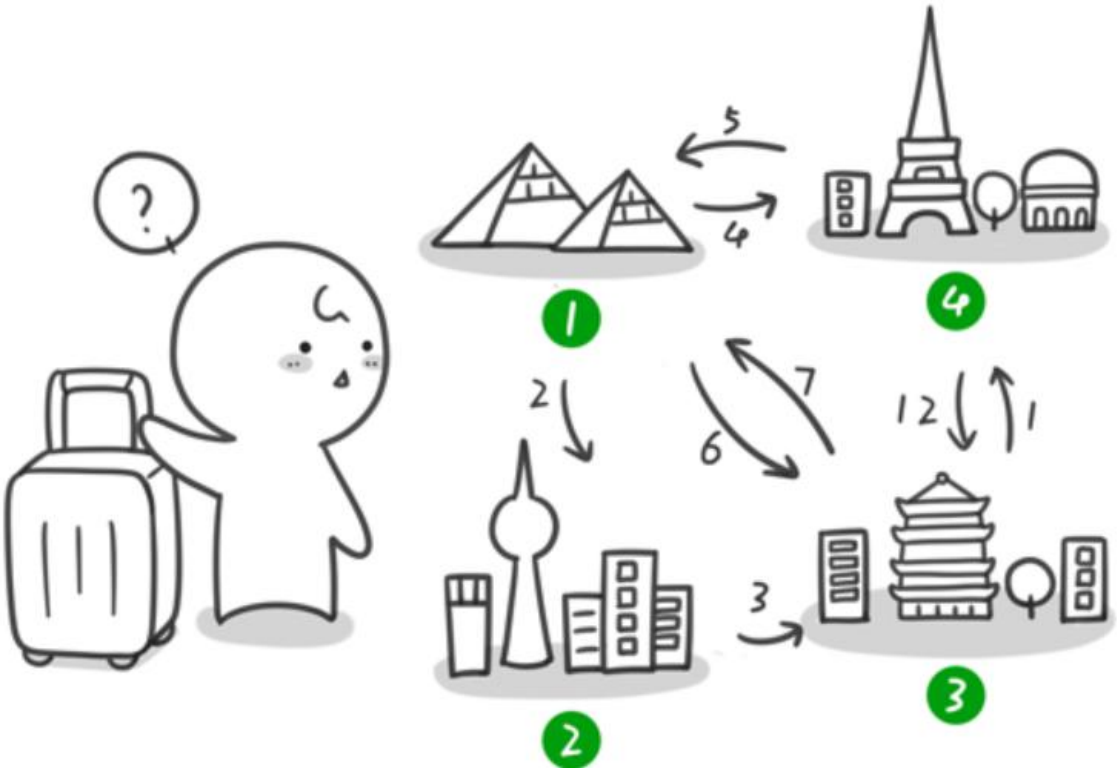
dis数组

v1	v2	v3	v4	v5	v6
0	∞	10	50	30	60

由于队列为空，队列循环结果，此时我们也得到了v1到各个顶点的最短路径的值了，它就是dis数组各个顶点对应的值

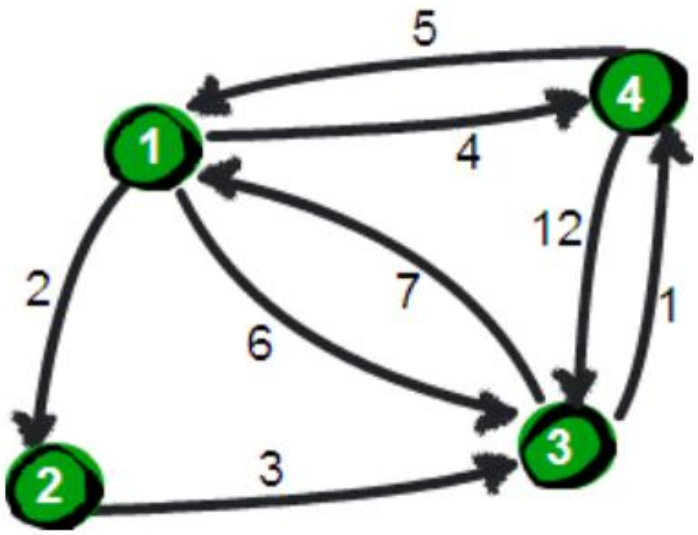
4 Floyd算法

小哼准备去一些城市旅游。有些城市之间有公路，有些城市之间则没有，如下图。为了节省经费以及方便计划旅程，小哼希望在出发之前知道任意两个城市之前的最短路程。



Floyd算法

Floyd 是解决任意两点间最短路径(称为多源最短路径问题)的经典算法，可以正确处理有向图或负权的最短路径问题。



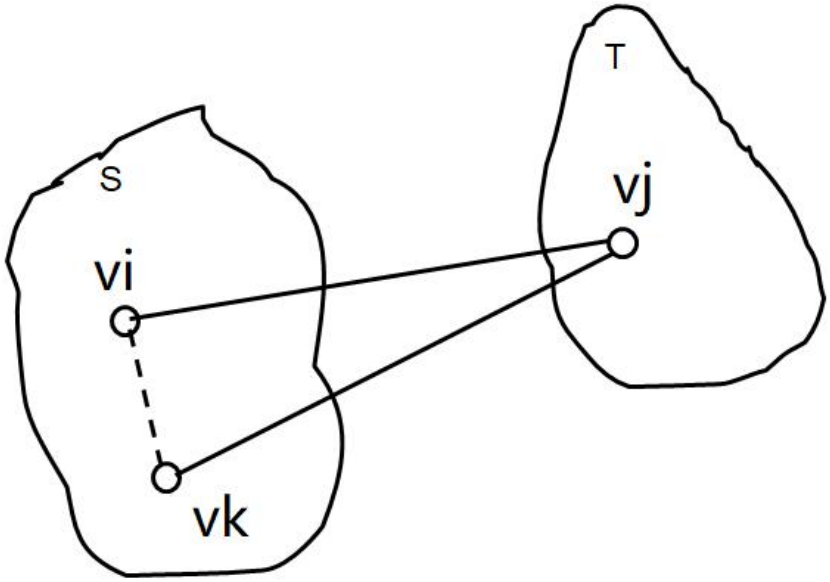
	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	∞	0	1
4	5	∞	12	0

算法分析

从任意节点i到任意节点j的最短路径不外乎2种可能：

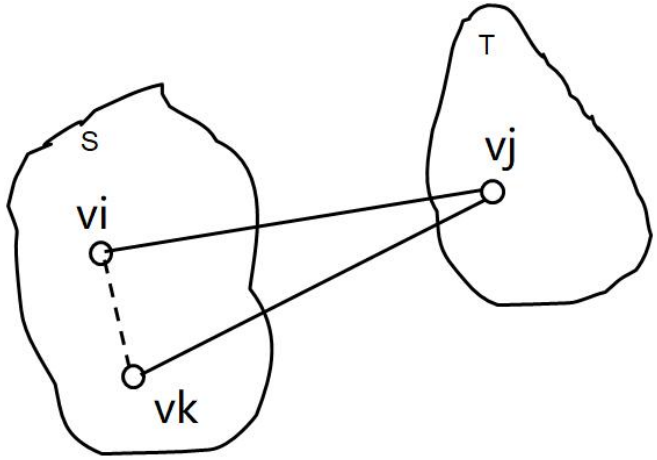
1) 直接从节点i到节点j。

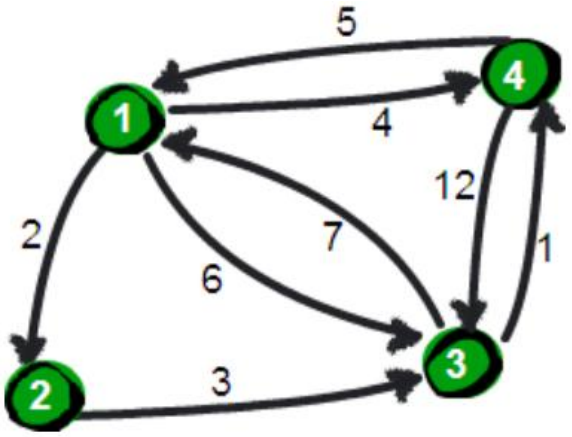
2) 从节点i经过若干个节点k到节点j。



可以证明vi到v_j的最短路径，要么是从vi到v_j的直接路径；要么是从v_i经某个顶点 v_k 再到v_j的路径。

Floyd-Warshall算法





递推公式：

初始：*dist* [i][j] = Edge[i][j]，v_i是源点

递推：*dist* [i][j] = min{ *dist* [i][j], *dist* [i][k] + dis[k][j] }, $v_k \in T$

其中v_u是当前dist[ ]最小的顶点。

算法的具体步骤

```
1  for(k=1;k<=n;k++)//经过中间K点
2      for(i=1;i<=n;i++) //源点
3      for(j=1;j<=n;j++)  //目标点
4      if(dis[i][j]>dis[i][k]+dis[k][j])
5          dis[i][j]=dis[i][k]+dis[k][j];
```

```
4 8
1 2 2
1 3 6
1 4 4
2 3 3
3 1 7
3 4 1
4 1 5
4 3 12
```

```
0    2    5    4
9    0    3    4
6    8    0    1
5    7   10    0
```

```
//Floyd算法核心语句
for(k=1;k<=n;k++)
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(e[i][j]>e[i][k]+e[k][j] )
    e[i][j]=e[i][k]+e[k][j];
//输出最终的结果
for(i=1;i<=n;i++) {
    for(j=1;j<=n;j++)
        printf("%7d",e[i][j]);
    printf("\n");
}
```

1 [2896]牛的旅行

农民John的农场里有很多牧区。有的路径连接一些特定的牧区。一片所有连通的牧区称为一个牧场。但是就目前而言，你能看到至少有两个牧区不连通。现在，John想在农场里添加一条路径（注意，恰好一条）。对这条路径有这样的限制：一个牧场的直径就是牧场中最远的两个牧区的距离（本题中所提到的所有距离指的都是最短的距离）。考虑如下的两个牧场，图1是有5个牧区的牧场，牧区用“*”表示，路径用直线表示。每一个牧区都有自己的坐标：

图1所示的牧场的直径大约是12.07106，最远的两个牧区是A和E，它们之间的最短路径是A-B-E。

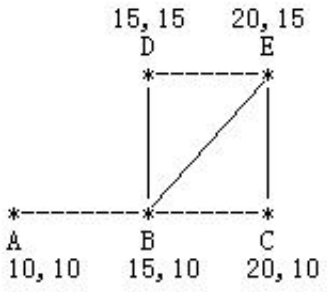


图1

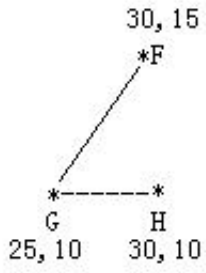


图2

[2896]牛的旅行

这两个牧场都在John的农场上。John将会在两个牧场中各选一个牧区，然后用一条路径连起来，使得连通后这个新的更大的牧场有最小的直径。注意，如果两条路径中途相交，我们不认为它们是连通的。只有两条路径在同一个牧区相交，我们才认为它们是连通的。

现在请你编程找出一条连接两个不同牧场的路径，使得连上这条路径后，这个更大的新牧场有最小的直径。

输入描述：

第 1 行：一个整数N (1 ≤ N ≤ 150), 表示牧区数；

第 2 到 N+1 行：每行两个整数X，Y (0 ≤ X，Y≤ 100000)，表示N个牧区的坐标。每个牧区的坐标都是不一样的。

第 N+2 行到第 2*N+1 行：每行包括N个数字 (0或1) 表示一个对称邻接矩阵。

例如，题目描述中的两个牧场的矩阵描述如下：

A B C D E F G H

A 0 1 0 0 0 0 0 0

B 1 0 1 1 1 0 0 0

C 0 1 0 0 1 0 0 0

D 0 1 0 0 1 0 0 0

E 0 1 1 1 0 0 0 0

F 0 0 0 0 0 0 1 0

G 0 0 0 0 0 1 0 1

H 0 0 0 0 0 0 1 0　输入数据中至少包括两个不连通的牧区。

输出描述：

每组输出只有一行，包括一个实数，表示所求答案。数字保留六位小数。

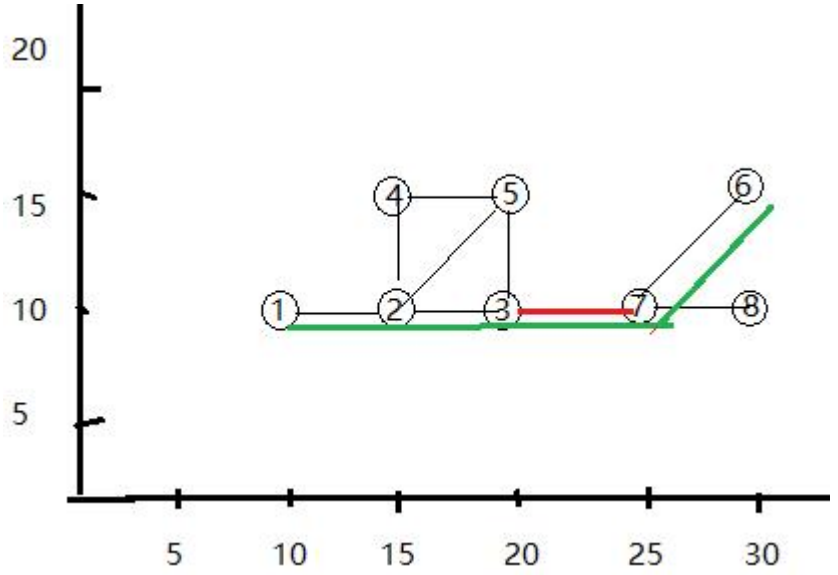
[2896]牛的旅行

样例输入：

8
10 10
15 10
20 10
15 15
20 15
30 15
25 10
30 10
01000000
10111000
01001000
01001000
01110000
00000010
00000101
00000010

样例输出：

22.071068



3和7之间添一条边
直径：1 2 3 6 7

分析

题意：在一个有n个节点的无向图中，有若干个联通块，并且至少有两个联通块之间是不连通的，节点称为牧区，联通块称为牧场，牧场的直径为这个联通块中两个节点之间路径长度的最大值，整张图的直径为各个联通块直径的最大值。

题目要求：我们添加一条边，使两个本来不连通的联通块能够连通。那么就产生了一个新的牧场和这个牧场的直径，我们需要使加入了一条边后的图的直径最小。

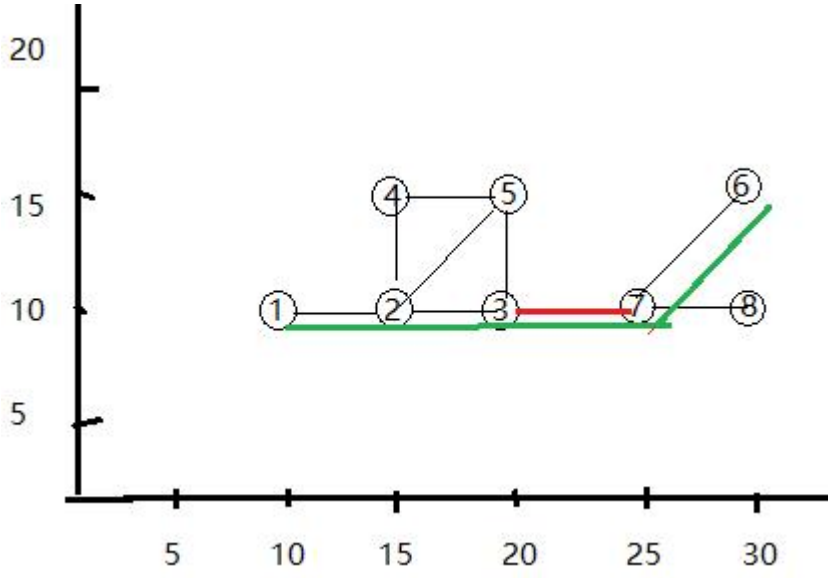
处理：我们先处理原图中各个联通块的直径，题目要求为最短距离，就可以用floyd算出各个联通节点之间的最小距离，再求以每个节点节点为端点的路径长度最大值。

再处理加入一条边后产生的直径（加入边的长度+一个节点为端点的路径最大值+另一个节点为端点的路径最大值）的最小值，最后在原图的直径与新产生的直径中取大的那个即为所求的直径最小值。

分析

1. 根据坐标算出边的权值（两节点之间的距离）
2. 用floyd算出任意两点之间的最短距离
3. 算出以每个点为一端的所有路径中最长距离
4. 算出添加一条边后产生的新直径的最小值（新增边的长度加上两个端点的路径长度最大值）

5取原有的最大直径与新直径的较大值



各个点为端点的路径长度最大值

1：12.071068　1 - > 5
2：7.071068　2 -> 5
3：10　3 -> 1或 3 -> 4
4：10　4 -> 1或 4 -> 3
5：12.071068　5 - >1
6：12.071068　6 - >7
7：7.0701068　7 - >6
8：12.071068　8 - >6

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N=155;
const double inf=0x3f3f3f3f; //无穷大
char g[N][N];  //存邻接矩阵（无向图）
int n;
double d[N][N],m[N]; //d 为任意两节点之间的距离，m为以每个节点为一端的所有路径中长度最大值
struct node{
    int x,y; //存每个点的坐标
}q[N];
double get(node a,node b){    //计算两个节点（有边直接相连）的距离
    double dx=a.x-b.x,dy=a.y-b.y;
    return sqrt(dx*dx+dy*dy);
}
int main(){
    cin>>n;
    for(int i=0;i<n;i++)
        cin>>q[i].x>>q[i].y; //输入每个节点的坐标
    for(int i=0;i<n;i++)
        cin>>g[i];  //输入邻接矩阵（即两个节点之间是否有边直接相连）
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            if(i!=j)  //若i=j，则为自己到自己，即距离为0（全局变量默认为0）
            {
                if(g[i][j]=='1') d[i][j]=get(q[i],q[j]); //‘1’表示两节点之间有边，则算出这两节点之间的距离
                else d[i][j]=inf;    //‘0’表示两点之间没有边直接相连，距离为无穷大
```

```
            if(i!=j)  //若i=j，则为自己到自己，即距离为0（全局变量默认为0）
            {
                if(g[i][j]=='1') d[i][j]=get(q[i],q[j]); //‘ 1’ 表示两节点之间有边，则算出这两节点之间的距离
                else d[i][j]=inf;    //‘ 0’ 表示两点之间没有边直接相连，距离为无穷大
            }
        }
    }
    for(int k=0;k<n;k++)  //floyd 算各个节点（联通的节点）之间的最短距离
       for(int i=0;i<n;i++)
          for(int j=0;j<n;j++)
             d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
    for(int i=0;i<n;i++)
       for(int j=0;j<n;j++)
          if(d[i][j]<inf)  //当i和j这两个节点之间联通时
            m[i]=max(m[i],d[i][j]); //求以i 节点为一端的所有路径中长度最大值 （另一端遍历除i之外的节点）
    double ans1=-1; //表示在原图中各个牧场的直径的最大值
    for(int i=0;i<n;i++)
       ans1=max(m[i],ans1);
    double ans2=inf; //表示添加一条边产生的新的牧场的直径 （求新产生直径的最小值）
    for(int i=0;i<n;i++)
       for(int j=0;j<n;j++)
          if(d[i][j]>=inf) //添加边的前提当然是这条边原来是不存在的（即两个节点之间不连通）
            ans2=min(ans2,get(q[i],q[j])+m[i]+m[j]);//添加后产生的直径为两个节点之间的直接距离加上以两个节点为一段的路径长度最大值
    double ans=max(ans1,ans2); // 原来的直径最大值与新牧场的直径取大的那个值
    printf("%f",ans); //输出小数点后六位
    return 0;
}
```

今天的课程结束啦……

下课了…
同学们再见！