最短路径问题

2017年3月10日　　17:06

1.问题引入

问题：从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——**最短路径**。解决最短路的问题有以下算法，**Dijkstra算法，Bellman-Ford算法，Floyd算法和SPFA算法，另外还有著名的启发式搜索算法A***.

2.Dijkstra算法

2.1 定义概览

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。**主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法，**该算法要求图中不存在负权边。算法复杂度O(n^2)

问题描述：**在无向图 G=(V,E) 中，假设每条边 E[i] 的长度为 w[i]，找到由顶点 V0 到其余各点的最短路径。（单源最短路径）**

2.2 算法描述

2.2.1 算法思想

算法思想：设G=(V,E)是一个带权有向图，**把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合**（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 ， 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），**第二组为其余未确定最短路径的顶点集合（用U表示）**，按最短路径长度的递增次序依次把第二组的顶点加入S中。**在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。**

2.2.2 算法步骤

a.初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则<u,v>正常有权值，若u不是v的出边邻接点，则<u,v>权值为∞。

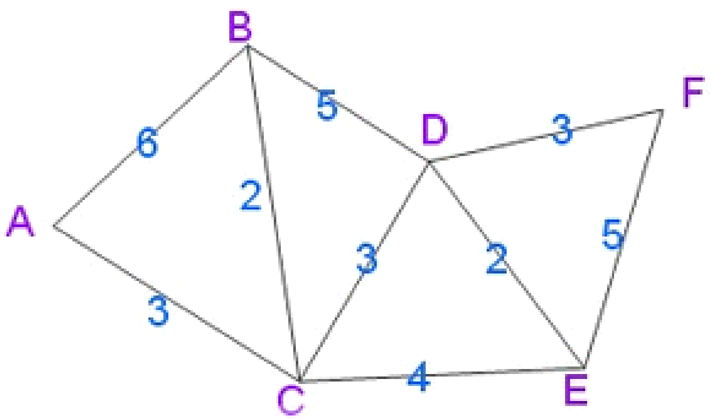
b.从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。

c.以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值为顶点k的距离加上边上的权。

d.重复步骤b和c直到所有顶点都包含在S中。

2.2.3 算法实例

先给出一个无向图



用Dijkstra算法找出以A为起点的单源最短路径步骤如下

步骤	S集合中	U集合中
1	选入A，此时S=<A> 此时最短路径A→A=0 以A为中间点，从A开始找	U=<B、C、D、E、F> A→B=6 A→C=3 A→其他U中的顶点=∞ 发现A→C=3权值为最短
2	选入C，此时S=<A，C> 此时最短路径A→A=0，A→C=3 以C为中间点，从A→C=3这条最短路径开始找	U=<B、D、E、F> A→C→B=5（比上面第一步的A→B=6要短） 此时到B权值为A→C→B=5 A→C→D=6 A→C→E=7 A→C→其他U中的顶点=∞ 发现A→C→B=5权值为最短
3	选入B，此时S=<A、C、B> 此时最短路径 A→A=0，A→C=3，A→C→B=5 以B为中间点，从A→C→B=5这条最短路径开始找	U=<D、E、F> A→C→B→D=10（比上面第二步的A→C→D=6要长） 此时到D权值更改为A→C→D=6 A→C→B→其他U中的顶点=∞ 发现A→C→D=6权值为最短
4	选入D，此时S=<A、C、B、D> 此时最短路径A→A=0，A→C=3，A→C→B=5，A→C→D=6 以D为中间点,从A→C→D这条最短路径开始找	U=<E、F> A→C→D→E=8（比上面第二步的A→C→E=7要长） 此时到E权值更改为A→C→E=7 A→C→D→F=9 发现A→C→E=7权值为最短
5	选入E，此时S=<A、C、B、D、E> 此时最短路径A→A=0，A→C=3，A→C→B=5，A→C→D=6 A→C→E=7 以E为中间点,从A→C→E=7这条最短路径开始找	U=<F> A→C→E→F=12（比上面第四步的A→C→D→F=9要长）此时到F权值更改为A→C→D→F=9 发现A→C→D→F=9权值为最短
6	选入F，此时S=<A、C、B、D、E、F> 此时最短路径A→A=0，A→C=3，A→C→B=5，A→C→D=6 A→C→E=7，A→C→D→F=9	U集合已空，查找完毕。

2.3 代码实现

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"
```

```
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXEDgE 20
#define MAXVEX 20
#define INFINITY 65535

typedef int Status;    /* Status是函数的类型,其值是函数结果状态代码，如OK等 */
typedef struct
{
    int vexs[MAXVEX];
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
}Mgraph;

typedef int Patharc[MAXVEX];               /* 用于存储最短路径下标的数组 */
typedef int ShortPathTable[MAXVEX];        /* 用于存储到各点最短路径的权值和 */

void CreateMgraph(Mgraph *g)
{
    int i, j;
    /* printf("请输入边数和顶点数:"); */
    g->numEdges=16;
    g->numVertexes=9;
    for (i = 0; i < g->numVertexes; i++)/* 初始化图 */
    {
        g->vexs[i]=i;
    }
    for (i = 0; i < g->numVertexes; i++)/* 初始化图 */
    {
        for ( j = 0; j < g->numVertexes; j++)
        {
            if (i==j)
                g->arc[i][j]=0;
            else
                g->arc[i][j] = g->arc[j][i] = INFINITY;
        }
    }
    g->arc[0][1]=1;
    g->arc[0][2]=5;
    g->arc[1][2]=3;
    g->arc[1][3]=7;
    g->arc[1][4]=5;
    g->arc[2][4]=1;
    g->arc[2][5]=7;
    g->arc[3][4]=2;
    g->arc[3][6]=3;
    g->arc[4][5]=3;
    g->arc[4][6]=6;
    g->arc[4][7]=9;
    g->arc[5][7]=5;
    g->arc[6][7]=2;
    g->arc[6][8]=7;
    g->arc[7][8]=4;
    for(i = 0; i < g->numVertexes; i++)
    {
        for(j = i; j < g->numVertexes; j++)
```

```
            {
                g->arc[j][i] =g->arc[i][j];
            }
        }
}

/* Dijkstra算法，求有向网g的v0顶点到其余顶点v的最短路径P[v]及带权长度D[v] */
/* P[v]的值为前驱顶点下标,D[v]表示v0到v的最短路径长度和 */
void ShortestPath_Dijkstra(Mgraph g, int v0, Patharc *P, ShortPathTable *D)
{
    int v,w,k,min;
    int final[MAXVEX];                    /* final[w]=1表示求得顶点v0至vw的最短路径 */

    /* 初始化数据 */
    for(v=0; v<g.numVertexes; v++)
    {
        final[v] = 0;                     /* 全部顶点初始化为未知最短路径状态 */
        (*D)[v] = g.arc[v0][v];           /* 将与v0点有连线的顶点加上权值 */
        (*P)[v] = 0;                      /* 初始化路径数组P为0 */
    }
    (*D)[v0] = 0;                         /* v0至v0路径为0 */
    final[v0] = 1;                        /* v0至v0不需要求路径 */

    /* 开始主循环，每次求得v0到某个v顶点的最短路径 */
    for(v=1; v<g.numVertexes; v++)
    {
        min=INFINITY;                     /* 当前所知离v0顶点的最近距离 */
        for(w=0; w<g.numVertexes; w++) /* 寻找离v0最近的顶点 */
        {
            if(!final[w] && (*D)[w]<min)
            {
                k=w;
                min = (*D)[w];            /* w顶点离v0顶点更近 */
            }
        }
        final[k] = 1;                     /* 将目前找到的最近的顶点置为1 */
        /* 修正当前最短路径及距离 */
        for(w=0; w<g.numVertexes; w++)
        {
            /* 如果经过v顶点的路径比现在这条路径的长度短的话 */
            if(!final[w] && (min+g.arc[k][w]<(*D)[w]))
            {
                /* 说明找到了更短的路径，修改D[w]和P[w] */
                (*D)[w] = min + g.arc[k][w]; /* 修改当前路径长度 */
                (*P)[w]=k;
            }
        }
    }
}

int main(void)
{
    int i,j,v0;
    Mgraph g;
    Patharc P;
    ShortPathTable D; /* 求某点到其余各点的最短路径 */
    v0=0;
```

```
    CreateMgraph(&g);

    ShortestPath_Dijkstra(g, v0, &P, &D);
    printf("最短路径倒序如下:\n");
    for(i=1;i<g.numVertexes;++i)
    {
        printf("v%d - v%d : ",v0,i);
        j=i;
        while(P[j]!=0)
        {
            printf("%d ",P[j]);
            j=P[j];
        }
        printf("\n");
    }
    printf("\n源点到各顶点的最短路径长度为:\n");
    for(i=1;i<g.numVertexes;++i)
        printf("v%d - v%d : %d \n",g.vexs[0],g.vexs[i],D[i]);
    return 0;
}
```

3.Floyed 算法

3.1 定义概览

Floyd-Warshall算法（Floyd-Warshall algorithm）是解决**任意两点间的最短路径的一种算法**，可以正确处理**有向图或负权**的最短路径问题，同时也被用于计算有向图的传递闭包。Floyd-Warshall算法的**时间复杂度为O(N3)，空间复杂度为O(N2)。**

3.2 算法描述

3.2.1 算法思想

Floyd算法是一个经典的动态规划算法。用通俗的语言来描述的话，首先我们的目标是寻找从点i到点j的最短路径。从动态规划的角度看问题，我们需要为这个目标重新做一个诠释（这个诠释正是动态规划最富创造力的精华所在）

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是直接从i到j，2是从i经过若干个节点k到j。所以，我们假设Dis(i,j)为节点i到节点j的最短路径的距离，**对于每一个节点k，我们检查Dis(i,k) + Dis(k,j) < Dis(i,j)是否成立**，如果成立，证明从i到k再到j的路径比i直接到j的路径短，我们**便设置Dis(i,j) = Dis(i,k) + Dis(k,j)**，这样一来，当我们遍历完所有节点k，Dis(i,j)中记录的便是i到j的最短路径的距离。

3.2.2 算法描述

a.从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。
b.对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

3.2.3 Floyd算法过程矩阵的计算----十字交叉法

方法：两条线，从左上角开始计算一直到右下角 如下所示
给出矩阵，其中矩阵A是邻接矩阵，而矩阵Path记录u,v两点之间最短路径所必须经过的点

$$A_{-1}=\begin{bmatrix}0 & 5 & \infty & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix} \qquad Path_{-1}=\begin{bmatrix}-1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\end{bmatrix}$$

相应计算方法如下：

（1）把 A_1 划去第 0 行第 0 列和对角线来计算 A_0

$$\begin{bmatrix}0 & 5 & \infty & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix}$$

不在三条线上的元素所在的 2 阶矩阵为：

$$\begin{bmatrix}0 & \infty\\ \infty & 4\end{bmatrix}, \begin{bmatrix}0 & 7\\ \infty & 2\end{bmatrix}, \begin{bmatrix}0 & 5\\ 3 & 3\end{bmatrix}, \begin{bmatrix}0 & 7\\ 3 & 2\end{bmatrix}, \begin{bmatrix}0 & 5\\ \infty & \infty\end{bmatrix}, \begin{bmatrix}0 & \infty\\ \infty & 1\end{bmatrix}$$

很容易就可以看出不在三条线上的 6 个元素都不发生改变，因此 $A_0 = A_{-1}$，$Path_0 = Path_{-1}$

(2) 把 A_0 划去第 1 行第 1 列和对角线来计算 A_1

$$\begin{bmatrix}0 & 5 & \infty & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix}$$

按上面所述的判断不在三条线上的元素是否需要发生改变，发上变化的元素用括号括起来了。。。

$$\begin{bmatrix}0 & 5 & \infty(9) & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix} \text{所以有 } A_1=\begin{bmatrix}0 & 5 & 9 & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix} \quad Path_1=\begin{bmatrix}-1 & -1 & 1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\end{bmatrix}$$

（3）把 A_1 划去第 2 行第 2 列和对角线来计算 A_2

$$\begin{bmatrix}0 & 5 & 9 & 7\\ \infty(7) & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty(4) & \infty(4) & 1 & 0\end{bmatrix} \text{所以有 } A_2=\begin{bmatrix}0 & 5 & 9 & 7\\ 7 & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ 4 & 4 & 1 & 0\end{bmatrix} \quad Path_2=\begin{bmatrix}-1 & -1 & 1 & -1\\ 2 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ 2 & 2 & -1 & -1\end{bmatrix}$$

（4）把 A_2 划去第 3 行第 3 列和对角线来计算 A_3

$$\begin{bmatrix}0 & 5 & 9(8) & 7\\ 7(6) & 0 & 4(3) & 2\\ 3 & 3 & 0 & 2\\ 4 & 4 & 1 & 0\end{bmatrix} \text{所以有 } A_3=\begin{bmatrix}0 & 5 & 8 & 7\\ 6 & 0 & 3 & 2\\ 3 & 3 & 0 & 2\\ 4 & 4 & 1 & 0\end{bmatrix} \quad Path_3=\begin{bmatrix}-1 & -1 & 3 & -1\\ 3 & -1 & 3 & -1\\ -1 & -1 & -1 & -1\\ 2 & 2 & -1 & -1\end{bmatrix}$$

3.3 代码实现

```
#include <iostream>
#include <string>
#include <stdio.h>
using namespace std;
```

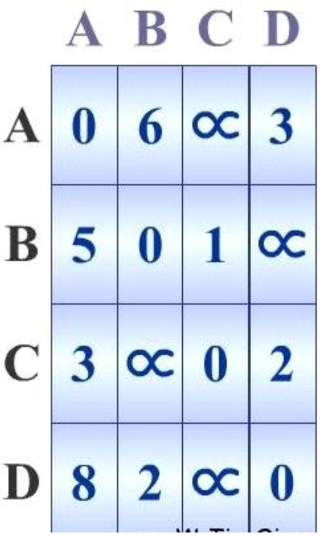
```
#define MaxVertexNum 100
#define INF 32767
typedef struct
{
    char vertex[MaxVertexNum];
    int edges[MaxVertexNum][MaxVertexNum];
    int n,e;
}MGraph;

void CreateMGraph(MGraph &G)
{
    int i,j,k,p;
    cout<<"请输入顶点数和边数:";
    cin>>G.n>>G.e;
    cout<<"请输入顶点元素:";
    for (i=0;i<G.n;i++)
    {
        cin>>G.vertex[i];
    }
    for (i=0;i<G.n;i++)
    {
        for (j=0;j<G.n;j++)
        {
            G.edges[i][j]=INF;
            if (i==j)
            {
                G.edges[i][j]=0;
            }
        }
    }
    for (k=0;k<G.e;k++)
    {
        cout<<"请输入第"<<k+1<<"条弧头弧尾序号和相应的权值:";
        cin>>i>>j>>p;
        G.edges[i][j]=p;
    }
}
void Dispath(int A[][MaxVertexNum],int path[][MaxVertexNum],int n);

void Floyd(MGraph G)
{
      int A[MaxVertexNum][MaxVertexNum],path[MaxVertexNum][MaxVertexNum];
      int i,j,k;
      for (i=0;i<G.n;i++)
      {
            for (j=0;j<G.n;j++)
            {
                  A[i][j]=G.edges[i][j];
                  path[i][j]=-1;
            }
      }
      for (k=0;k<G.n;k++)
      {
            for (i=0;i<G.n;i++)
            {
                  for (j=0;j<G.n;j++)
                  {
                        if (A[i][j]>A[i][k]+A[k][j])
```

```
                    {
                        A[i][j]=A[i][k]+A[k][j];
                        path[i][j]=k;
                    }
                }
            }
        }
        Dispath(A,path,G.n);
}

void Ppath(int path[][MaxVertexNum],int i,int j)
{
        int k;
        k=path[i][j];
        if (k==-1)
        {
            return;
        }
        Ppath(path,i,k);
        printf("%d,",k);
        Ppath(path,k,j);
}

void Dispath(int A[][MaxVertexNum],int path[][MaxVertexNum],int n)
{
        int i,j;
        for (i=0;i<n;i++)
        {
            for (j=0;j<n;j++)
            {
                if (A[i][j]==INF)
                {
                    if (i!=j)
                    {
                        printf("从%d到%d没有路径\n",i,j);
                    }
                }
                else
                {
                    printf("  从%d到%d=>路径长度:%d路径:",i,j,A[i][j]);
                    printf("%d,",i);
                    Ppath(path,i,j);
                    printf("%d\n",j);
                }
            }
        }
}

int main()
{
        freopen("input2.txt", "r", stdin);
        MGraph G;
        CreateMGraph(G);
        Floyd(G);
        return 0;
}
```

代码的几点说明

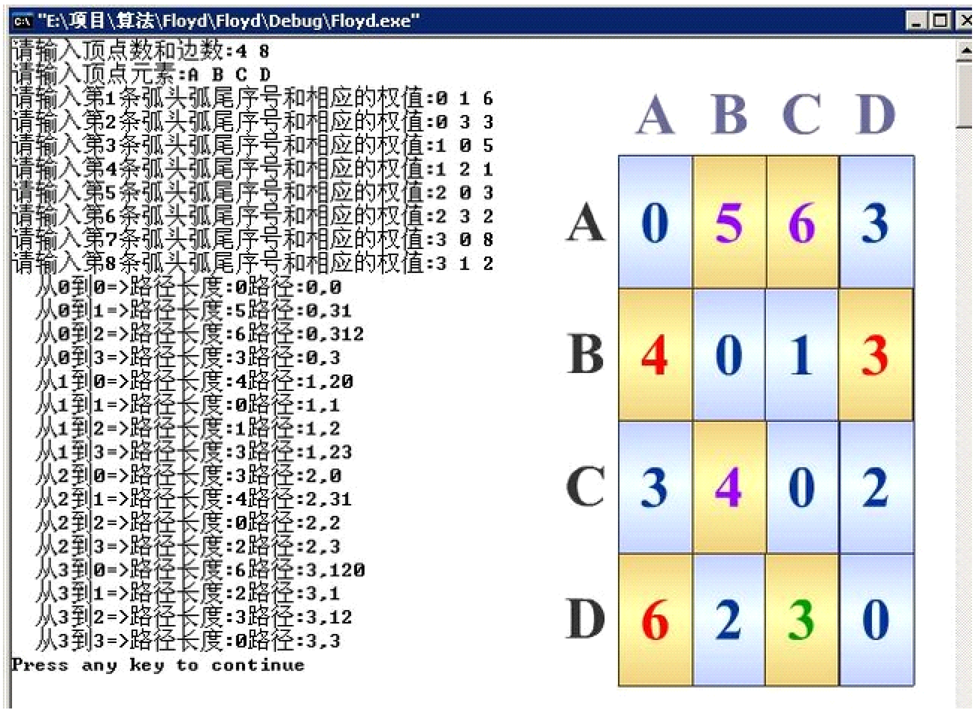
- A[][]数组初始化为各顶点间的原本距离，最后存储各顶点间的最短距离。
- path[][]数组保存最短路径，与当前迭代的次数有关。初始化都为-1，表示没有中间顶点。在求A[i][j]过程中，path[i][j]存放从顶点vi到顶点vj的中间顶点编号不大于k的最短路径上前一个结点的编号。在算法结束时，由二维数组path的值回溯，可以得到从顶点vi到顶点vj的最短路径。

实例:初始化A[][]数组为如下，即有向图的邻接矩阵。



	A	B	C	D
A	0	6	∝	3
B	5	0	1	∝
C	3	∝	0	2
D	8	2	∝	0

运行结果：



```
"E:\项目\算法\Floyd\Floyd\Debug\Floyd.exe"
请输入顶点数和边数:4 8
请输入顶点元素:A B C D
请输入第1条弧头弧尾序号和相应的权值:0 1 6
请输入第2条弧头弧尾序号和相应的权值:0 3 3
请输入第3条弧头弧尾序号和相应的权值:1 0 5
请输入第4条弧头弧尾序号和相应的权值:1 2 1
请输入第5条弧头弧尾序号和相应的权值:2 0 3
请输入第6条弧头弧尾序号和相应的权值:2 3 2
请输入第7条弧头弧尾序号和相应的权值:3 0 8
请输入第8条弧头弧尾序号和相应的权值:3 1 2
  从0到0=>路径长度:0路径:0,0
  从0到1=>路径长度:5路径:0,31
  从0到2=>路径长度:6路径:0,312
  从0到3=>路径长度:3路径:0,3
  从1到0=>路径长度:4路径:1,20
  从1到1=>路径长度:0路径:1,1
  从1到2=>路径长度:1路径:1,2
  从1到3=>路径长度:3路径:1,23
  从2到0=>路径长度:3路径:2,0
  从2到1=>路径长度:4路径:2,31
  从2到2=>路径长度:0路径:2,2
  从2到3=>路径长度:2路径:2,3
  从3到0=>路径长度:6路径:3,120
  从3到1=>路径长度:2路径:3,1
  从3到2=>路径长度:3路径:3,12
  从3到3=>路径长度:0路径:3,3
Press any key to continue
```

	A	B	C	D
A	0	5	6	3
B	4	0	1	3
C	3	4	0	2
D	6	2	3	0

4.Bellman-Ford算法

4.1 简介

Dijkstra算法是处理单源最短路径的有效算法，但它局限于边的权值非负的情况，若图中出现权值为负的边，Dijkstra算法就会失效，求出的最短路径就可能是错的。这时候，就需要使用其他的算法来求解最短路径，Bellman-Ford算法就是其中最常用的一个。该算法由美国数学家理查德•贝尔曼（Richard Bellman, 动态规划的提出者）和小莱斯特•福特（Lester Ford）发明。

适用条件&范围

1. 单源最短路径(从源点s到其它所有顶点v);
2. 有向图&无向图(无向图可以看作(u,v),(v,u)同属于边集E的有向图);
3. 边权可正可负(如有负权回路输出错误提示);
4. 差分约束系统;

4.2算法描述

4.2.1 算法思想

给定图G(V, E)（其中V、E分别为图G的顶点集与边集），源点s，

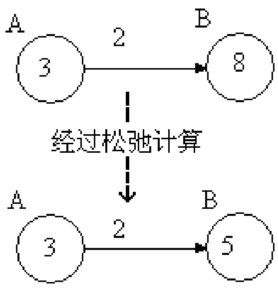
- 数组Distant[i]记录从源点s到顶点i的路径长度，初始化数组Distant[n]为maxint, Distant[s]为0；
- 以下操作循环执行至多n-1次，n为顶点数：对于每一条边e(u, v)，如果Distant[u] + w(u, v) < Distant[v]，则令Distant[v] = Distant[u]+w(u, v)。w(u, v)为边e(u,v)的权值；

  若上述操作没有对Distant进行更新，说明最短路径已经查找完毕，或者部分点不可达，跳出循环。否则执行下次循环；
- 为了检测图中是否存在负环路，即权值之和小于0的环路。对于每一条边e(u, v)，如果存在Distant[u] + w(u, v) < Distant[v]的边，则图中存在负环路，即是说该图无法求出单源最短路径。否则数组Distant[n]中记录的就是源点s到各顶点的最短路径长度。

可知，Bellman-Ford算法寻找单源最短路径的时间复杂度为O(V*E).

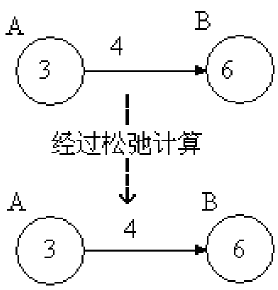
4.2.2 算法流程

首先介绍一下松弛计算。如下图：



松弛计算之前，点B的值是8，但是点A的值加上边上的权重2，得到5，比点B的值（8）小，所以，点B的值减小为5。这个过程的意义是，找到了一条通向B点更短的路线，且该路线是先经过点A，然后通过权重为2的边，到达点B。

当然，如果出现一下情况



则不会修改点B的值，因为3＋4>6。

Bellman－Ford算法可以大致分为三个部分

第一，初始化所有点。每一个点保存一个值，表示从原点到达这个点的距离，将原点的值设为0，其它的点的值设为无穷大（表示不可达）。

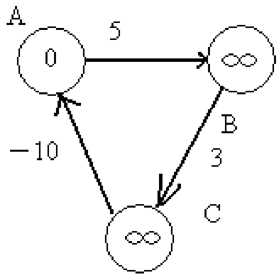
第二，进行循环，循环下标为从1到n－1（n等于图中点的个数）。在循环内部，遍历所有的边，进行松弛计算。

第三，遍历途中所有的边（edge（u，v）），判断是否存在这样情况：

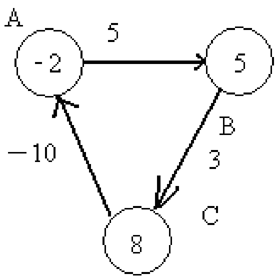
d（v） > d (u) + w(u,v)

则返回false，表示途中存在从源点可达的权为负的回路。

之所以需要第三部分的原因，是因为，如果存在从源点可达的权为负的回路。则 应为无法收敛而导致不能求出最短路径。考虑如下的图：

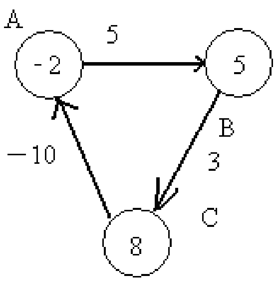


经过第一次遍历后，点B的值变为5，点C的值变为8，这时，注意权重为－10的边，这条边的存在，导致点A的值变为－2。（8＋ －10＝－2）



第二次遍历后，点B的值变为3，点C变为6，点A变为－4。正是因为有一条负边在回路中，导致每次遍历后，各个点的值不断变小。

在回过来看一下bellman－ford算法的第三部分，遍历所有边，检查是否存在d（v） > d (u) + w(u,v)。因为第二部分循环的次数是定长的，所以如果存在无法收敛的情况，则肯定能够在第三部分中检查出来。比如



此时，点A的值为－2，点B的值为5，边AB的权重为5，5 > -2 + 5. 检查出来这条边没有收敛。

所以，Bellman—Ford算法可以解决图中有权为负数的边的单源最短路径问题。

4.2代码实现

```
/*
* About:  Bellman-Ford算法
* Author: Tanky Woo
* Blog:   www.WuTianqi.com
*/

#include <iostream>
using namespace std;
const int maxnum = 100;
const int maxint = 99999;

// 边，
typedef struct Edge{
	int u, v;    // 起点，重点
	int weight;  // 边的权值
}Edge;

Edge edge[maxnum];     // 保存边的值
int  dist[maxnum];     // 结点到源点最小距离

int nodenum, edgenum, source;    // 结点数，边数，源点

// 初始化图
void init()
{
	// 输入结点数，边数，源点
	cin >> nodenum >> edgenum >> source;
	for(int i=1; i<=nodenum; ++i)
		dist[i] = maxint;
	dist[source] = 0;
	for(int i=1; i<=edgenum; ++i)
	{
		cin >> edge[i].u >> edge[i].v >> edge[i].weight;
		if(edge[i].u == source)          //注意这里设置初始情况
			dist[edge[i].v] = edge[i].weight;
	}
}

// 松弛计算
void relax(int u, int v, int weight)
{
	if(dist[v] > dist[u] + weight)
		dist[v] = dist[u] + weight;
}

bool Bellman_Ford()
{
	for(int i=1; i<=nodenum-1; ++i)
		for(int j=1; j<=edgenum; ++j)
			relax(edge[j].u, edge[j].v, edge[j].weight);
	bool flag = 1;
	// 判断是否有负环路
	for(int i=1; i<=edgenum; ++i)
		if(dist[edge[i].v] > dist[edge[i].u] + edge[i].weight)
```

```
            {
                flag = 0;
                break;
            }
        return flag;
}
int main()
{
        //freopen("input3.txt", "r", stdin);
    init();
        if(Bellman_Ford())
            for(int i = 1 ;i <= nodenum; i++)
                cout << dist[i] << endl;
        return 0;
}
```

补充：

考虑：为什么要循环V-1次？

答：因为最短路径肯定是个简单路径，不可能包含回路的，如果包含回路，且回路的权值和为正的，那么去掉这个回路，可以得到更短的路径如果回路的权值是负的，那么肯定没有解了.图有n个点，又不能有回路,所以最短路径最多n-1边，又因为每次循环，至少relax一边，所以最多n-1次就行了

5.SPFA算法

5.1 RELAX(松弛)简介

在介绍SPFA之前需要先理解算法中的关键步骤，**RELAX(松弛)操作**，简介如下：

单源最短路径算法中使用了**松弛（relaxation）**操作。对于每个顶点v∈V，都设置一个属性d[v]，用来描述从源点s到v的最短路径上权值的上界，称为**最短路径估计（shortest-path estimate）。π[v]代表S到v的当前最短路径中v点之前的一个点的编号**,我们用下面的Θ(V)时间的过程来对最短路径估计和前趋进行初始化。

```
INITIALIZE-SINGLE-SOURCE(G,s)
1    for each vertex v∈V[G]
2       do d[v]←∞
3          π[v]←NIL
4    d[s]←0
```

经过初始化以后，对所有v∈V，π[v]=NIL，对v∈V-{s}，有d[s]=0以及d[v]=∞。

在松弛一条边(u,v)的过程中，要测试是否可以通过u，对迄今找到的v的最短路径进行改进；如果可以改进的话，则更新d[v]和π[v]。一次松弛操作可以减小最短路径估计的值d[v]，并更新v的前趋域π[v](S到v的当前最短路径中v点之前的一个点的编号)。下面的伪代码对边(u,v)进行了一步松弛操作。

```
RELAX(u, v, w)
1    if(d[v]>d[u]+w(u,v))
2       then d[v]←d[u]+w(u,v)
3            π[v]←u
```

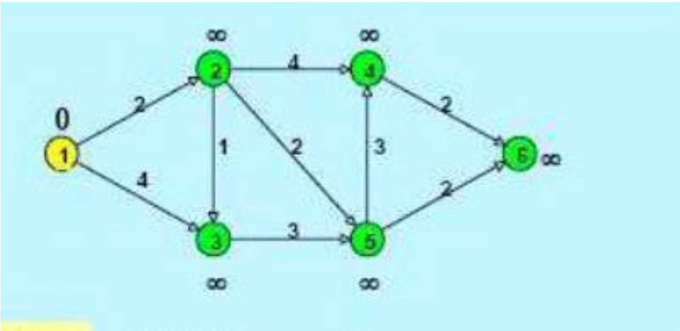
每个单源最短路径算法中都会调用INITIALIZE-SINGLE-SOURCE，然后重复对边进行松弛的过程。另外，松弛是改变最短路径和前趋的唯一方式。各个单源最短路径算法间区别在于对每条边进行松弛操作的次数，以及对边执行松弛操作的次序有所不同。在Dijkstra算法以及关于有向无回路图的最短路径算法中，对每条边执行一次松弛操作。在Bellman-Ford算法中，每条边要执行多次松弛操作。

5.2 SPFA简介

SPFA(Shortest Path Faster Algorithm)是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。
解决存在负环的图的单源最短路径，bellman-ford**算法**是比较经典的一个，但是大家都知道，这个算法的效率很低，因为它只知道要求单源最短路，至多做|v|（j图的结点数）次松弛操作。

西南交通大学段凡丁1994年发明了算法SPFA，很大程度上优化了bellman-ford算法。SPFA算法的精妙之处在于不是盲目的做松弛操作，而是用一个队列保存当前做了松弛操作的结点。只要队列不空，就可以继续从队列里面取点，做松弛操作。

举例说明：



当前源点1在队列里面，于是我们取了1结点来做对图进行松弛操作，显然这个时候2,3结点的距离更新了，入了队列，我们假设他们没入队列,即现在队列已经空了，那么就没有必要继续做松弛操作，因为源点1要到其他结点必须经过2或3结点。

SPFA的基本思想：

算法大致流程是用一个队列来进行维护。 初始时将源加入队列。 每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，如果该点没有在队列中，则将其入队。 直到队列为空时算法结束。

判断有无负环：如果某个点进入队列的次数超过V次则存在负环（**SPFA无法处理带负环的图**）。
SPFA算法有两个优化算法 SLF 和 LLL： SLF：Small Label First 策略，设要加入的节点是j，队首元素为i，若dist(j)<dist(i)，则将j插入队首，否则插入队尾。 LLL：Large Label Last 策略，设队首元素为i，队列中所有dist值的平均值为x，若dist(i)>x则将i插入到队尾，查找下一元素，直到找到某一i使得dist(i)<=x，则将i出对进行松弛操作。 SLF 可使速度提高 15 ~ 20%；SLF + LLL 可提高约 50%。 在实际的应用中SPFA的算法时间效率不是很稳定，为了避免最坏情况的出现，通常使用效率更加稳定的Dijkstra算法。个人觉得LLL优化每次要求平均值，不太好，为了简单，我们可以直接用**C++** STL里面的优先队列来进行SLF优化。

用优先队列来进行SLF优化代码：

```
#include <iostream>
#include <cstring>
#include <queue>
using namespace std;
const int INF = 0x3ffffff;
```

```
const int MAX = 100;
int map[MAX][MAX];
int dis[MAX];
bool vis[MAX];
int num[MAX];//记录每个结点入队的次数
struct cmp
{
     bool operator()(int x,int y)
     {
          return x>y;
     }
};
bool SPFA(int s0,int n)
{
     priority_queue<int,vector<int>,cmp> q;
     memset(vis,false,sizeof(vis));
     memset(num,0,sizeof(num));
     for(int i=0;i<n;i++)
          dis[i] = INF;
     dis[s0] = 0;
     q.push(s0);
     vis[s0] = true;
     num[s0]++;
     while(!q.empty())
     {
          int p = q.top();
          q.pop();
          for(int i=0;i<n;i++)
          {
               if(dis[p]+map[p][i]<dis[i])
               {
                    dis[i] = dis[p]+map[p][i];
                    if(!vis[i])
                    {
                         q.push(i);
                         num[i]++;
                         if(num[i]>n)//存在负环
                         {
                              return false;
                         }
                         vis[i]=true;
                    }
               }
          }
          vis[p] = false;
     }
     return true;
}
```