安徽工业大学

管理科学与工程学院

《运筹学》

实验指导书

专业：信息管理与信息系统

班级：息 171

姓名：钱继鹏

学号：179094299

教师：胡火群/潘瑞林

日期：2019.7.5

目录

1. 求最小生成树.....

2. 求单源顶点最短路径.....

3. 求网络最大流.....

4. 线性规划问题求解.....

实验一 求最小生成树

（实验学时：4 学时； 实验类型：D 设计研究性 ）

一、实验目的

（1） 掌握图的最小生成树的概念

（2） 学会抽象数据概念，建立数学、计算机模型

（3） 掌握最小生成树的 Prim 算法

（4） 学会使用编程语言实现 Prim 算法求解最小生成树

二、实验内容（主要指解决什么问题）

（1） 了解图在计算机里的存储结构

（2） 用邻接矩阵建立图的存储

（3） 用 C++语言实现 Prim 算法

（4） 完善代码，提供良好的用户体验

三、实验原理和方法

算法原理：每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点 s 开始，逐渐长大覆盖整个连通网的所有顶点。

1. 图的所有顶点集合为 V；初始令集合 u={s},v=V−uu={s},v=V−u;

2. 在两个集合 u,v 能够组成的边中，选择一条代价最小的边(u0,v0)，加入到最小生成树中，并把 v0 并入到集合 u 中。

3. 重复上述步骤，直到最小生成树有 n-1 条边或者 n 个顶点为止。

四、实验设备、工具、平台或软件名称

开发环境：

操作系统：Ubuntu 16.04 x64

编程语言：C++

编译器：GCC 5.4.0

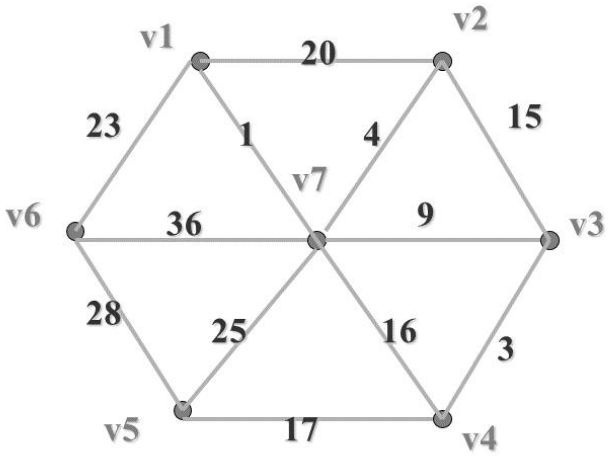
调试器：GDB 7.11.1

编辑器：qtCreator 5.11.1

依赖库：Boost(C++)

五、实验步骤（根据实验目的具体写出可操作性的实验步骤）

1. 问题分析



最小生成树就是权值最小的极小连通子图，求解可以利用已有算法 Prim 算法，通过建立模型，将其实现一遍即可。

2. 模型建立

图在计算机中的存储方式，有邻接矩阵、邻接表等，为计算方便，在此使用邻接矩阵，建立一个二维邻接矩阵，再对这个邻接矩阵实现 Prim 算法。

3. 程序架构设计

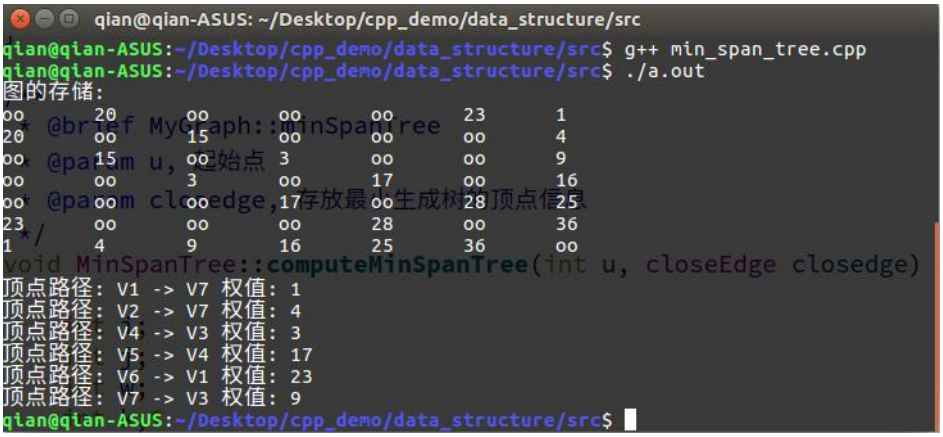
采用面向对象的编程思想，设计一个 MyGraph 类全部成员访问权限设为 public，用以实现图的存储，接着设计一个 MinSpanTree 类来 public 继承 MyGraph 类，使得 MinSpanTree 也拥有了 MyGraph 类的成员属性。

接着就可以在 MinSpanTree 类中设计一个成员函数来实现 Prim 算法了。

具体代码见附录。

4. 程序测试与调试

以本题数据为例，进行测试调试，结果如下：



六、实验要求及注意事项

（1） 编写代码要遵循良好的代码规范，要有必要的注释，变量命名要规范。

（2） 算法设计要满足可行性与唯一性

（3） 建立有好的人机交互界面

七、思考题（根据具体实验项目可以无此项）

(1) 还有没有其他算法了？

有，Kruskal 算法，与 Prim 算法不同的是，Kruskal 算法是采用“加边”法，把原始图看成 N 个独立的顶点，不断地选取当前权值最小的边，连线，加入到生成树中。

八、 附录（代码）

1. MyGraph.h(只在这里写一遍，下面的实验还会用到)

```
#ifndef MYGRAPH_H
#define MYGRAPH_H

#include <iostream>
#include <boost/lexical_cast.hpp>
using namespace std;

const int MAXVERTEXNUM = 100;
const int MAXWEIGHT = 100;
class MyGraph   //虚基类
{
//private:
public:
    int M_vertexNum;
    int M_edgeNum;
    int M_vertexs[MAXVERTEXNUM];                // 顶点序号
    int M_edges[MAXVERTEXNUM][MAXVERTEXNUM];    // 权值
public:
    MyGraph()
    { }
    virtual ~MyGraph()
```

```
        { }

        /**
         * @brief initGraph, virtual method in super class!
         * @note you should overwrite this method in drivered class!
         */
        virtual void initGraph() = 0;

        /**
         * @brief showOriginalGraph
         */
        void showOriginalGraph() {
            int i;
            int j;
            int count = 0 ;
            cout << "图的存储: " << endl;
            for ( i=0; i<M_vertexNum; i++ )
                for ( j=0; j<M_vertexNum; j++ )
                {
                    count ++;
                    string tmp =
boost::lexical_cast<string>( M_edges[j][i] );

                    if ( M_edges[j][i] == MAXWEIGHT )
                        tmp = "oo";
                    cout << tmp << "\t";
                    if ( count%7 == 0 )
                        cout << endl;
                }
            cout << endl;
            return;
        }
    };
    #endif // MYGRAPH_H
```

2. MinSpanTree.h

```
#ifndef MIN_SPAN_TREE_H
#define MIN_SPAN_TREE_H
#include "mygraph.h"

// prim 算法求最小生成树的辅助数组
typedef struct array{
    int adjvertex;  // 某顶点与已经构件好的部分生成树的顶点之间 权值最小的顶点
    int lowcost;    // .....的最小权值
}closeEdge[MAXVERTEXNUM];

// min span tree class
class MinSpanTree:public MyGraph
{
public:
    MinSpanTree()
    { }
    virtual ~MinSpanTree()
    { }
    /**
     * @brief initGraph, virtual method in super class!
     */
    virtual void initGraph();
    void computeMinSpanTree(int u, closeEdge closedge);
};
#endif // MIN_SPAN_TREE_H
```

3. MinSpanTree.cpp

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

```
/**
 * @brief MinSpanTree::initGraph
 */
void MinSpanTree::initGraph() {
    M_vertexNum = 7;
    M_edgeNum = 12;
    int i;
    int j;
    for ( i = 0; i < 7; ++i) {
        M_vertexs[i] = i;
    }

    for ( i=0; i<7; i++ )
        for ( j=0; j<7; j++ )
            M_edges[i][j] = MAXWEIGHT;

    M_edges[0][1] = 20;
    M_edges[1][2] = 15;
    M_edges[2][3] = 3;
    M_edges[3][4] = 17;
    M_edges[4][5] = 28;
    M_edges[5][0] = 23;
    M_edges[0][6] = 1;
    M_edges[1][6] = 4;
    M_edges[2][6] = 9;
    M_edges[3][6] = 16;
    M_edges[4][6] = 25;
    M_edges[5][6] = 36;

    M_edges[1][0] = 20;
    M_edges[2][1] = 15;
    M_edges[3][2] = 3;
    M_edges[4][3] = 17;
    M_edges[5][4] = 28;
```

```
        M_edges[0][5] = 23;
        M_edges[6][0] = 1;
        M_edges[6][1] = 4;
        M_edges[6][2] = 9;
        M_edges[6][3] = 16;
        M_edges[6][4] = 25;
        M_edges[6][5] = 36;

        return;
    }

    /**
     * @brief MyGraph::minSpanTree
     * @param u, 起始点
     * @param closedge, 存放最小生成树的顶点信息
     */
    void MinSpanTree::computeMinSpanTree(int u, closeEdge closedge)
{

        int i;
        int j;
        int w;
        int k;
        // 辅助数组初始化
        for ( i=0; i<M_vertexNum; i++ ) {
            if ( i != u ) {
                closedge[i].adjvertex = u;
                closedge[i].lowcost = M_edges[u][i];
            }
        }
        closedge[u].lowcost = 0;

        // 选择其余的 n-1 的顶点
        for (i = 0;  i< M_vertexNum-1; i++) {
```

```
            w = MAXWEIGHT;
            for ( j=0; j<M_vertexNum; j++ ) {
                if ( closedge[j].lowcost != 0 && closedge[j].lowcost
< w ) {
                    w = closedge[j].lowcost;
                    k = j;
                }
            }

            closedge[k].lowcost = 0;    // 第 k 顶点并入 U 集合
            // 修改辅助数组
            for ( j=0; j<M_vertexNum; j++ ) {
                if ( M_edges[k][j] < closedge[j].lowcost ) {
                    closedge[j].adjvertex = k;
                    closedge[j].lowcost = M_edges[k][j];
                }
            }
        }

        // 打印每条边
        string start;
        string next;
        for ( i=0; i<M_vertexNum; i++ ) {
            if ( i != u ) {

                // handle the vertex info
                switch (i) {
                case 0:
                    start = "V1";
                    break;
                case 1:
                    start = "V2";
                    break;
                case 2:
```

```
            start = "V3";
            break;
        case 3:
            start = "V4";
            break;
        case 4:
            start = "V5";
            break;
        case 5:
            start = "V6";
            break;
        case 6:
            start = "V7";
            break;
        default:
            start = "NONE";
        }

        switch (closedge[i].adjvertex) {
        case 0:
            next = "V1";
            break;
        case 1:
            next = "V2";
            break;
        case 2:
            next = "V3";
            break;
        case 3:
            next = "V4";
            break;
        case 4:
            next = "V5";
            break;
```

```
                case 5:
                    next = "V6";
                    break;
                case 6:
                    next = "V7";
                    break;
                default:
                    next = "NONE";
                }
                cout << "顶点路径: " << start << " -> " << next << "
权值: " << M_edges[i][closedge[i].adjvertex] << endl;
            }
        }

        return;
    }

    /**
     * @brief main
     * @return
     */
    int main() {
        MinSpanTree S;
        closeEdge closedge;
        S.initGraph();
        S.showOriginalGraph();
        S.computeMinSpanTree(2, closedge);

        return 0;
    }
```

实验二　求单源顶点最短路径

（实验学时：4 学时； 实验类型：D 设计研究性 ）

一、实验目的

（1） 掌握最短路径的概念

（2） 学会抽象数据概念，建立数学、计算机模型

（3） 掌握单源顶点最短路径的 Dijkstra 算法

（4） 学会使用编程语言实现 Dijkstra 算法求解最短路径

二、实验内容（主要指解决什么问题）

（1） 了解图在计算机里的存储结构

（2） 用邻接矩阵建立图的存储

（3） 用 C++语言实现 Dijkstra 算法

（4） 完善代码，提供良好的用户体验

三、实验原理和方法

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. 算法步骤：

（1） 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∞。

（2） 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）。

（3） 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，修改后的距离值的顶点 k 的距离加上边上的权。

（4） 重复步骤 b 和 c 直到所有顶点都包含在 S 中。

四、实验设备、工具、平台或软件名称

开发环境：

操作系统：Ubuntu 16.04 x64

编程语言：C++

编译器：GCC 5.4.0

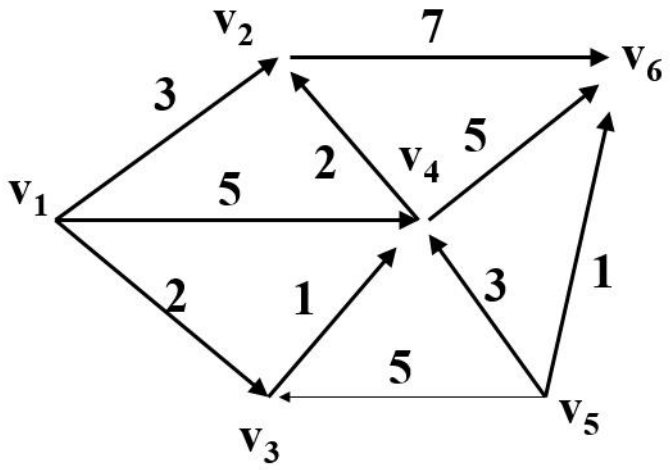
调试器：GDB 7.11.1

编辑器：qtCreator 5.11.1

依赖库：Boost(C++)

五、实验步骤（根据实验目的具体写出可操作性的实验步骤）

1. 问题分析



找出一条从 V1 到 V6 的最短路径，可以用 Dijkstra 算法求解，建立图的存储，将 Dijkstra 算法用编程语言实现一遍即可。

2. 模型建立

图在计算机中的存储方式，有邻接矩阵、邻接表等，为计算方便，在此使用邻接矩阵，建立一个二维邻接矩阵，再对这个邻接矩阵实现 DIjkstra 算法。

3. 程序架构设计

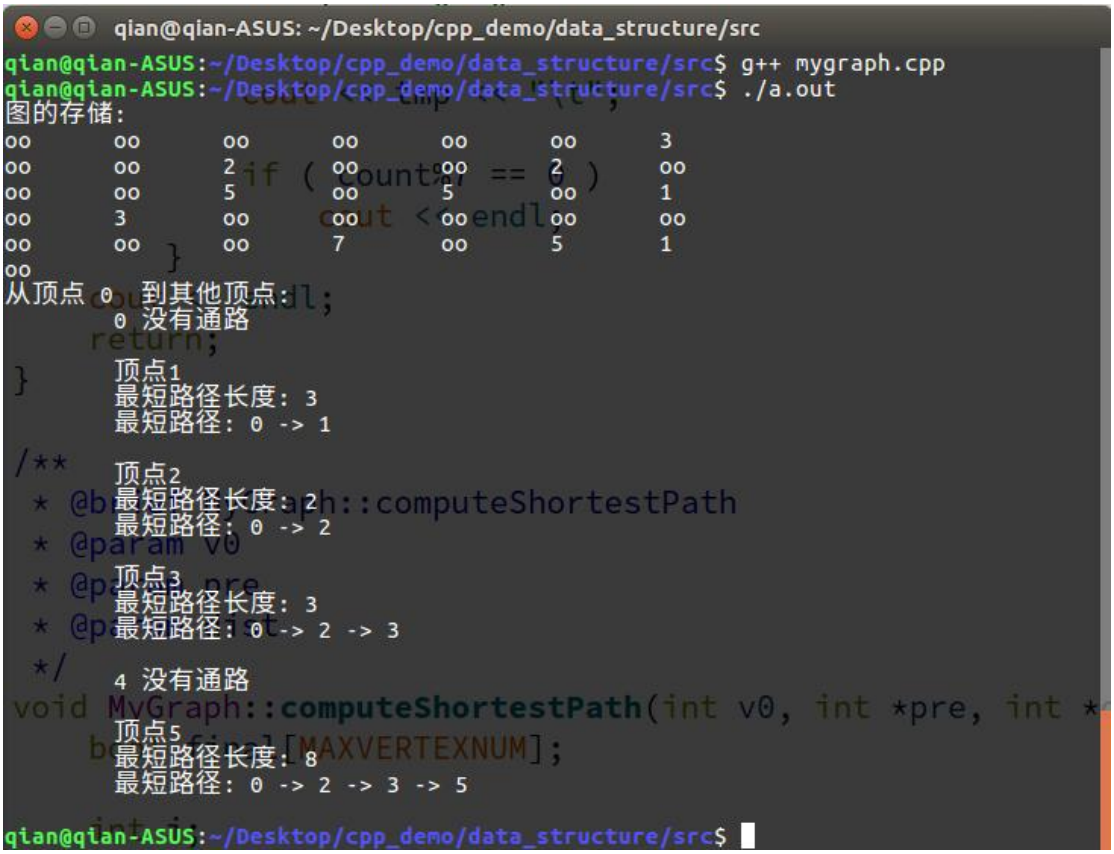
采用面向对象的编程思想，设计一个 MyGraph 类全部成员访问权限设为 public，用以实现图的存储，接着设计一个 MinPath 类来 public 继承 MyGraph 类，使得 MinPath 也拥有了 MyGraph 类的成员属性。

接着就可以在 MinPath 类中设计一个成员函数来实现 Dijkstra 算法了。

具体代码见附录。

4. 程序测试与调试

以本题数据为例，进行测试调试，结果如下：



六、实验要求及注意事项

（1） 编写代码要遵循良好的代码规范，要有必要的注释，变量命名要规范。

（2） 掌握单源最短路径的手算算法和计算机算法 Dijsktra 算法

（3） 注意是无向图还是有向图

七、思考题（根据具体实验项目可以无此项）

（1） 还有没有其他算法了？

有，Floyed 算法，简单粗暴，时间复杂度较高，这里我没有采用。

（2） 如果是无向图怎么办？

在建立矩阵时再追加一条反向边权值，表示双向连通，即为无向的。

八、 附录（代码）

1. MinPath.h

```
#ifndef MIN_PATH_H
#define MIN_PATH_H
/**
  * dijkstra to compute the shortest path
  */
#include "mygraph.h"
class MinPath:public MyGraph
{
public:
    MinPath()
    { }
    virtual ~MinPath()
    { }

    virtual void initGraph();
    void computeShortestPath(int v0, int *pre, int *dist);
    void showShortestPath(int v0, int *pre, int *dist);
};
#endif // MIN_PATH_H
```

2. MinPath.cpp

```
#include <iostream>
#include <stack>
#include <boost/lexical_cast.hpp>
#include "min_path.h"
using namespace std;

/**
 * @brief MinPath::initGraph
 * @note the start vertex must start with '0'
 * @todo fix the vertex info(automatically to certain question)
 */
void MinPath::initGraph() {
    M_vertexNum = 6;
    M_edgeNum = 10;
```

```
        int i;
        int j;
        for ( i=0; i<M_vertexNum; i++ ) {
            this->M_vertexs[i] = i;
        }
        for ( i=0; i<M_vertexNum; i++ )
            for ( j=0; j<M_vertexNum; j++ )
                this->M_edges[i][j] = MAXWEIGHT;

        M_edges[0][1] = 3;
        M_edges[0][2] = 2;
        M_edges[0][3] = 5;
        M_edges[2][3] = 1;
        M_edges[3][1] = 2;
        M_edges[1][5] = 7;
        M_edges[4][2] = 5;
        M_edges[3][5] = 5;
        M_edges[4][3] = 3;
        M_edges[4][5] = 1;
        return;
    }

    /**
     * @brief MinPath::computeShortestPath
     * @param v0, the start vertex
     * @param pre, array to store the current vertex's pirror vertex
     * @param dist, array to store the current vertex's weight to the
start vertex
     */
    void MinPath::computeShortestPath(int v0, int *pre, int *dist) {
        bool final[MAXVERTEXNUM];

        int i;
        int w;
```

```
        int v;
        int current_tmp_min;
        // 初始化
        for ( v=0; v<=M_vertexNum-1; v++ ) {
            final[v] = false;
            dist[v] = M_edges[v0][v];       // 如果不直接连通，
dist[v]就是MAXWEIGHT,否则是相应的权值，我称此时dist[v] 为 "估计值
"
            pre[v]=-1;      // 所有的顶点都无前驱，置pre数组为 -1

            if ( dist[v] < MAXWEIGHT ) // v 到 v0 (直接)连通
                pre[v] = v0;
        }

        // 开始时V0属于S集合，默认已经找到最短路径
        dist[v0] = 0;
        final[v0] = true;       // (final[i] = true 相当于把i 加入
S集合)

        // main loop
        // 寻找其余6个节点
        for ( i=1; i<M_vertexNum+1; i++ ) {
            v = -1;     // ---> if (v==-1)
            current_tmp_min = MAXWEIGHT;    // 当前距V0路径最短的
权值 ,初始化为不连通状态

            // 寻找当前离V0最近的顶点 V
            // 第一次main loop 下，current_tmp_min 变化情况：dist[1]
-> dist[2]
            for ( w=0; w<M_vertexNum; w++ ) {
                if ( final[w] == false && dist[w] < current_tmp_min )
{  // w还没找到最短路径，并且d[w]比当前min 还要小
                    v = w;                          // 更新
                    current_tmp_min = dist[v];
```

```
                }
            }
            if ( v == -1 )                          // 所有与 V0 相通的
点都找到了最短路径(不满足 line 108 的 if )，则退出 main loop
                break;

            final[v] = true;                // 把 V 加入 S 集合

            // 更新当前最短路径及距离(V 作为中间点)
            for ( w=0; w<M_vertexNum; w++ ) {
                if ( final[w] == false &&
(current_tmp_min+(M_edges[v][w]) < dist[w]) )  // v0 -> v -> w 比 v0
-> w 短
                {
                    // 第一次 main loop, dist[1] 更新成 4, 2 作为 1
的前驱
                    dist[w] = current_tmp_min + (M_edges[v][w]);
// 更新最短路径长度, 此时 dist[w]为 "确定值"
                    pre[w] = v;     // v 作为 w 的前驱顶点
                }
            }
        }   //end main loop
        return;
    }

    /**
     * @brief MinPath::showShortestPath
     * @param v0
     * @param pre
     * @param dist
     */
    void MinPath::showShortestPath(int v0, int *pre, int *dist) {
        int v;
        int i;
```

```
        stack<int> s;
        cout << "从顶点 " << v0 << "  到其他顶点: " << endl;

        for ( v=0; v<M_vertexNum; v++ ) {
            if ( pre[v] == -1 )   //v0 到 v 不通
            {
                cout << "\t" << v << " 没有通路" << endl << endl;
                continue;
            }
            cout <<  "\t 顶点" << v << endl;
            cout << "\t 最短路径长度: " << dist[v] << endl;
            i = v;
            while ( pre[i] != -1 ) {
                //cout << "前驱: " << pre[i] << endl;
                s.push(pre[i]);             // 入栈，交换顺序

                i = pre[i];
            }
            cout << "\t 最短路径: ";
            for ( unsigned long size = s.size(); size > 0; size -- )
{
                cout << s.top() << " -> ";
                s.pop();
            }
            cout << v << endl << endl;
        }
        return;
    }

    /**
     * @brief main
     * @return
     */
    int main() {
```

```
        MinPath G;
        int pre[MAXVERTEXNUM];
        int dist[MAXVERTEXNUM];
        G.initGraph();
        G.showOriginalGraph();
        G.computeShortestPath(0, pre, dist);
        G.showShortestPath(0, pre, dist);
        return 0;
    }
```

实验三　求网络最大流

（实验学时：4 学时； 实验类型：D 设计研究性 ）

一、实验目的

（1） 掌握最大流的概念

（2） 学会抽象数据概念，建立数学、计算机模型

（3） 掌握 Edmonds-Karp 算法

（4） 学会使用编程语言实现 Edmonds-Karp 算法求解网络最大流

二、实验内容（主要指解决什么问题）

（1） 了解图在计算机里的存储结构

（2） 用邻接矩阵建立图的存储

（3） 用 C++语言实现 Edmonds-Karp 算法

（4） 完善代码，提供良好的用户体验

三、实验原理和方法

1. 名词解释：

(1) 残量网络

为了更方便算法的实现，一般根据原网络定义一个残量网络。其中 r(u,v)为残量网络的容量。

r(u,v) = c(u,v) - f(u,v)

通俗地讲：就是对于某一条边（也称弧），还能再有多少流量经过。

（2）增广路

在残量网络中的一条从 s 通往 t 的路径，其中任意一条弧(u,v)，都有 r[u,v]>0

（3）增广路算法

每次用 DFS 找一条最短的增广路径，然后沿着这条路径修改流量值（实际修改的是残量网络的边权）。当没有增广路时，算法停止，此时的流就是最大流

2. 算法思想：

求最大流的过程，就是不断找到一条源到汇的路径，若有，找出增广路径上每一段[容量-流量]的最小值 delta，然后构建残余网络，再在残余网络上寻找新的路径，使总流量增加。然后形成新的残余网络，再寻找新路径.....直到某个残余网络上找不到从源到汇的路径为止，最大流就算出来了

四、实验设备、工具、平台或软件名称

开发环境：

操作系统：Ubuntu 16.04 x64

编程语言：C++

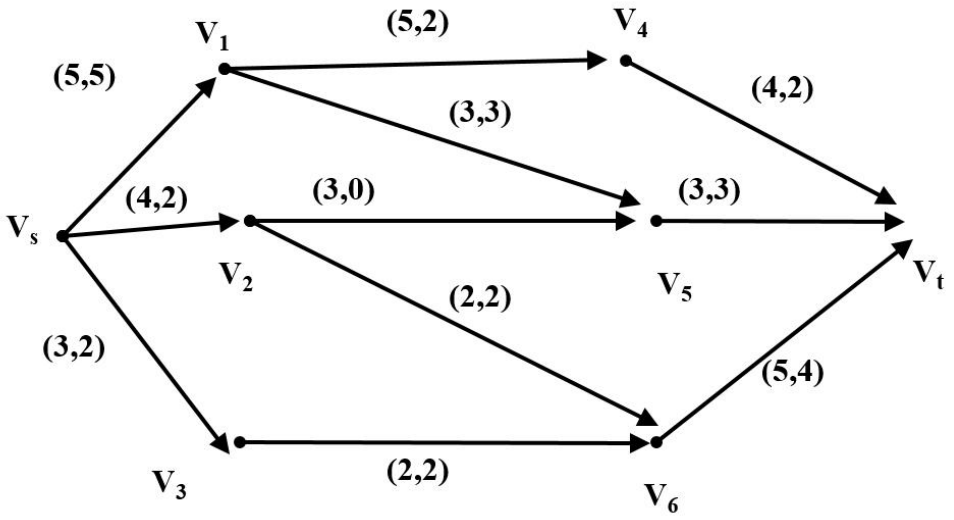
编译器：GCC 5.4.0

调试器：GDB 7.11.1

编辑器：qtCreator 5.11.1

五、实验步骤（根据实验目的具体写出可操作性的实验步骤）

1. 问题分析



2. 模型建立

图在计算机中的存储方式，有邻接矩阵、邻接表等，为计算方便，在此使用邻接矩阵，建立一个二维邻接矩阵，再对这个邻接矩阵实现算法。

3. 程序架构设计

采用 C 风格编写，命令行输入的方式

4. 程序测试与调试

以本题数据为例，进行测试调试，结果如下：

六、实验要求及注意事项

(1) 编写代码要遵循良好的代码规范，要有必要的注释，变量命名要规范。

(2) 熟练掌握网路最大流及其相关概念。

(3) 了解 EK 算法，求解网络最大流。

七、 附录（代码）

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
#include <iostream>
using namespace std;
const int INF=0x7fffffff;
```

```
queue <int> q;
int n,m,x,y,s,t,g[201][201],pre[201],flow[201],maxflow;
//g 邻接矩阵存图，pre 增广路径中每个点的前驱，flow 源点到这个点的流量


/**
 * @brief bfs
 * @param s
 * @param t
 * @return
 */
int bfs(int s,int t)
{
    while (!q.empty()) q.pop();
    for (int i=1; i<=n; i++) pre[i]=-1;
    pre[s]=0;
    q.push(s);
    flow[s]=INF;
    while (!q.empty())
    {
        int x=q.front();
        q.pop();
        if (x==t) break;
        for (int i=1; i<=n; i++)
            //EK 一次只找一个增广路
            if (g[x][i]>0 && pre[i]==-1)
            {
                pre[i]=x;
                flow[i]=min(flow[x],g[x][i]);
                q.push(i);
            }
    }
    if (pre[t]==-1) return -1;
```

```
    else return flow[t];
}

//increase 为增广的流量
void EK(int s,int t)
{
    int increase=0;
    while ((increase=bfs(s,t))!=-1)//这里的括号加错了！Tle
    {//迭代
        int k=t;
        while (k!=s)
        {
            int last=pre[k];//从后往前找路径
            g[last][k]-=increase;
            g[k][last]+=increase;
            k=last;
        }
        maxflow+=increase;
    }
}

/**
 * @brief main
 * @return
 */
int main()
{
    cout << "输入边数、顶点数: "<< endl;
    scanf("%d %d",&m,&n);
    cout << "输入起始点、终点、最大流量: " << endl;
    for (int i=1; i<=m; i++)
    {
        int z;
        scanf("%d %d %d",&x,&y,&z);
```

```
            g[x][y]+=z;//此处不可直接输入，要+=
        }
        EK(1,n);
        printf("最大流: %d\n",maxflow);
        return 0;
    }
```

实验四 线性规划问题求解

（实验学时：4 学时； 实验类型：C 综合性）

一、实验目的

（1） 学会将实际问题转化为数学模型，建立模型思想

（2） 掌握线性规划问题的单纯型解法

二、实验内容（主要指解决什么问题）

（1） 解决实际线性规划问题

（2） 单纯型表进行迭代

（3） 求出最优解

三、实验原理和方法

（1） 模型建立

（2） 化为标准型

（3） 确立初始可行基，机那里单纯形表

（4） 进行基变换，确立换入换出变量

（5） 交换

（6） 继续迭代

四、实验设备、工具、平台或软件名称

手工运算、无需其他工具

五、实验步骤（根据实验目的具体写出可操作性的实验步骤）

1. 问题分析

某公司生产铝框架玻璃门和木框玻璃门两种产品，它的下属工厂 1 生产

铝框，工厂 2 生产木框，工厂 3 生产玻璃并进行组装。生产单位产品 1 需工厂 1 的 1 个工时，工厂 3 的 3 个工时，可获利 3 元；生产单位产品 2 需工厂 2 的 2 个工时，工厂 3 的 3 个工时，可获利 5 元。现公司调整生产，得到如下剩余生产能力：工厂 1 可用工时为 4，工厂 2 可用工时为 12，工厂 3 可用工时为 18.制定两种产品的生产计划，使获得的利润最大。

2. 具体求解

（1） 按照题意，建立数学模型

设产品 1 生产 x1 件，产品 2 生产 x2 件，则构建模型为

$$\text{Max } z = 3X1 + 5X2$$

$$\begin{cases} X1 <= 4 \\ 2X2 <= 12 \\ 3X1 + 3X2 <= 18 \\ X1,X2,X3 >= 0 \end{cases}$$

（2） 通过加松弛变量、人工变量等方法，将模型化为标准型

$$\text{Max } z = 3X1 + 5X2 + 0X3 + 0X4 + 0X5$$

$$\begin{cases} X1 + X3 = 4 \\ 2X2 + X4 = 12 \\ 3X1 + 3X2 + X5 = 18 \\ X1,X2,X3,X4,X5 >= 0 \end{cases}$$

（3） 确定初始可行基，建立单纯形法

Cj->			3	5	0	0	0
Cb	Xb	b	X1	X2	X3	X4	X5
0	X3	4	1	0	1	0	0
0	X4	12	0	2	0	1	0
0	X5	18	3	3	0	0	1

则可得初始可行基 X(0)=(0,0,4,12,18),此时 z=0；

（4） 进行基变换，确定换入变量和换出变量

Cj->			3	5	0	0	0	θ i+
Cb	Xb	b	X1	X2	X3	X4	X5	
0	X3	4	1	0	1	0	0	-

0	X4	12	0		0	1	0	6
0	X5	18	3	3	0	0	1	6
Cj-Zj			3	5	0	0	0	

（5） 将换入变量的列化成标准形式

Cj->			3	5	0	0	0	θi+
Cb	Xb	b	X1	X2	X3	X4	X5	
0	X3	4	1	0	1	0	0	0
5	X2	6	0	1	0	0.5	0	-
0	X5	0		0	0	-1.5	1	0
Cj-Zj			3	0	0	-2.5	0	

（6） 继续进行迭代

Cj->			3	5	0	0	0	θi+
Cb	Xb	b	X1	X2	X3	X4	X5	
0	X3	4	0	0	1	0.5	-1/3	-
5	X2	6	0	1	0	0.5	0	-
3	X1	0	1	0	0	-0.5	1/3	-
Cj-Zj			0	0	0	-1	-1	

由检验数可知，此时已经达到最优解，此时解为 A（1）=（0，6，0，0，0）T,得最大值 Z = 30

六、实验要求及注意事项

（1） 计算要仔细，防止出错

（2） 理解检验数的计算方法，换入变量和换出变量的确认方法

（3） 记住单纯形表的格式