							0

实验课程	数据结构实验	成绩	
实验项目	实验三　图的操作与实现	指导老师	

一、 实验目的：

1、领会图的两种主要存储结构和图的基本运算算法设计；

2、领会图的两种遍历算法；

3、领会 Prim 算法求带权连通图中最小生成树的过程和相关算法设计；

4、掌握深度优先遍历和广度优先遍历算法在图路径搜索问题中的应用；

5、深入掌握图遍历算法在求解实际问题中的应用。

二、 使用仪器、器材

微机一台

操作系统：WinXP

编程软件：C/C++编程软件

三、 实验内容及原理

1、教材 P310 实验题 1：实现图的邻接矩阵和邻接表的存储

编写一个程序 graph.cpp，设计带权图的邻接矩阵与邻接表的创建和输出运算，并在此基础上设计一个主程序 exp8-1.cpp 完成以下功能。

（1）建立如图 8.54 所示的有向图 G 的邻接矩阵，并输出之。

（2）建立如图 8.54 所示的有向图 G 的邻接表，并输出之。

（3）销毁图 G 的邻接表。

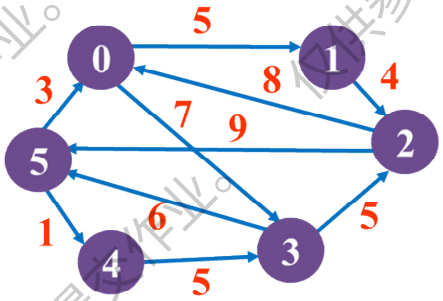


图 8.54 一个带权有向图

2、教材 P310 实验题 2：实现图的遍历算法

编写一个程序 travsal.cpp 实现图的两种遍历运算，并在此基础上设计一个程序 exp8-2.cpp 完成以下功能。

（1） 输出如图 8.54 的有向图 G 从顶点 0 开始的深度优先遍历序列（递归算法）。

（2） 输出如图 8.54 的有向图 G 从顶点 0 开始的深度优先遍历算法(非递归算法)。

（3） 输出如图 8.54 的有向图 G 从顶点 0 开始的广度优先遍历序列。

3、教材 P311 实验题 5：采用 Prim 算法求最小生成树

编写一个程序 exp8-5.cpp，实现求带权连通图中最小生成树的 Prim 算法，如图 8.55 所示的带权连通图 G，输出从顶点 0 出发的一棵最小生成树。

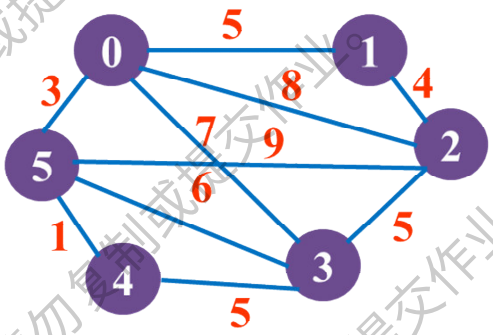


图 8.55 一个带权连通图

4、教材 P311 实验题 10：求有向图的简单路径

编写一个程序 exp8-10.cpp，设计相关算法完成以下功能。

（1） 输出如图 8.56 的有向图 G 从顶点 5 到顶点 2 的所有简单路径。

（2） 输出如图 8.56 的有向图 G 从顶点 5 到顶点 2 的所有长度为 3 的简单路径。

（3） 输出如图 8.56 的有向图 G 从顶点 5 到顶点 2 的最短路径。

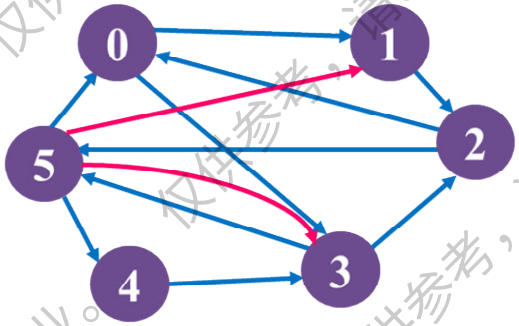


图 8.56 一个有向图

5、教材 P313 实验题 14：用图搜索方法求解如图 3.28（教材 P119）的迷宫问题（也可以自建迷宫）

编写一个程序 exp8-14.cpp，完成以下功能。

（1） 建立一个迷宫对应的邻接表表示。

（2） 采用深度优先遍历算法输出从入口（1，1）到出口（M,N）的所有迷宫路径。

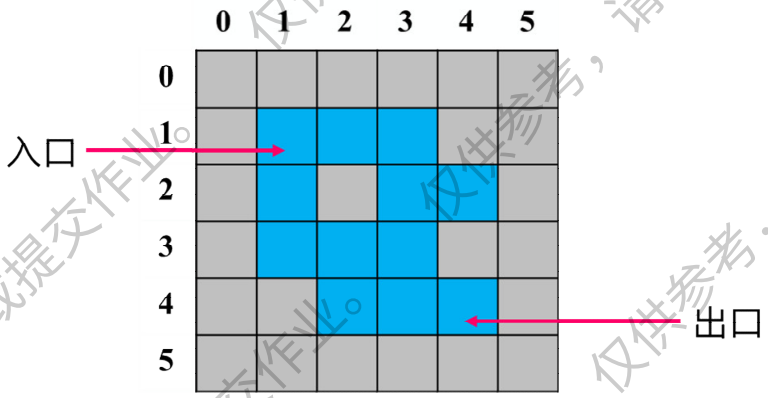


图 3.28 迷宫示意图

四、 实验过程原始数据记录

第 1 题：

// graph.h

```
#define MaxVertex 100          // 顶点数目的最大值
#define INFINITY 2147483647    // 无穷
typedef int VertexType;        // 顶点的数据类型
typedef int EdgeType;          // 带权图边权值的数据类型

// -------------- 邻接矩阵表示法 --------------
class MGraph
{
public:
    MGraph(VertexType* Vex, int n, int e);
    void PrintMGraph();

private:
    VertexType Vex[MaxVertex];                  // 顶点表
    EdgeType Edge[MaxVertex][MaxVertex];        // 邻接矩阵，边表
    int VexNum;                                 // 当前顶点数
    int ArcNum;                                 // 当前弧数
};

// -------------- 邻接表表示法 --------------
// 边表结点
typedef struct ArcNode
{
    int adjvex;             // 邻接点域
    EdgeType weight;        // 边的权重
```

```
    struct ArcNode* next;     // 指针域
}ArcNode;
// 顶点表结点
typedef struct VertxNode
{
    VertexType data;     // 顶点域
    ArcNode* first;      // 边表的头指针
}VNode, AdjList[MaxVertex];
// 邻接表储存的图
class ALGraph
{
public:
    ALGraph(VertexType* Vex, int n, int e);
    ~ALGraph();
    void PrintALG();

private:
    AdjList vertices;       // 邻接表
    int VexNum;             // 当前顶点数
    int ArcNum;             // 当前弧数

};
```

// graph.cpp

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

// -------------- 邻接矩阵表示法 --------------
MGraph::MGraph(VertexType* V, int n, int e)
{
    int val1, val2, weight;
    // 根据V[]初始化顶点信息
    for (int j = 0; j < n; j++)
        this->Vex[j] = j;
    // 初始化图的顶点数和弧数
    this->VexNum = n;
    this->ArcNum = e;
    // 初始化图邻接矩阵
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            this->Edge[i][k] = INFINITY;
    // 输入图的边，构造邻接矩阵
```

```
    cout << "请输入（格式如下，Tab分隔）：" << endl;
    cout << "顶点1\t顶点2\t权值" << endl;
    for (int j = 0; j < e; j++)
    {
        cin >> val1 >> val2 >> weight;
        this->Edge[val1][val2] = weight;
    }
}
void MGraph::PrintMGraph()
{
    cout << endl << "输出图的邻接矩阵如下：" << endl;
    for (int i = 0; i < this->VexNum; i++)
    {
        for (int k = 0; k < this->VexNum; k++)
        {
            if (this->Edge[i][k] == INFINITY)
                cout << "∞\t";
            else
                cout << this->Edge[i][k] << "\t";
        }
        cout << endl;
    }
}


// -------------- 邻接表表示法 --------------
ALGraph::ALGraph(VertexType* Vex, int n, int e)
{
    int val1, val2, val3;
    this->ArcNum = e;
    this->VexNum = n;
    // 初始化邻接表
    for (int i = 0; i < n; i++)
    {
        this->vertices[i].data = Vex[i];
        this->vertices[i].first = NULL;
    }
    // 输入图的边，构造邻接表
    cout << "请输入（格式如下，Tab分隔）：" << endl;
    cout << "顶点1\t顶点2\t权值" << endl;
    for (int j = 0; j < e; j++)
    {
        cin >> val1 >> val2 >> val3;
        ArcNode* s = new ArcNode;
```

```
            s->adjvex = val2;
            s->weight = val3;
            // 前插法
            s->next = vertices[val1].first;
            vertices[val1].first = s;
        }
    }
    ALGraph::~ALGraph()
    {
        ArcNode* q = NULL;  // 前驱工作指针
        ArcNode* p = NULL;  // 后驱工作指针
        for (int i = 0; i < VexNum; i++)
        {
            p = q = vertices[i].first;
            while (p)
            {
                q = p;
                p = p->next;
                delete q;
            }
            vertices[i].data = -1;
        }
        VexNum = 0;
        ArcNum = 0;
        cout << "销毁已完成！" << endl;
    }
    void ALGraph::PrintALG()
    {
        ArcNode* p = NULL;      // 工作指针
        cout << endl << "输出图的邻接表如下：" << endl;
        for (int i = 0; i < VexNum; i++)
        {
            cout << vertices[i].data << "： ";
            p = vertices[i].first;
            while (p)
            {
                cout << p->adjvex << "(" << p->weight << ")  ";
                p = p->next;
            }
            cout << endl;
        }

    }
```

// exp8-1.cpp

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

int main()
{
    cout << "（1）建立如图所示的有向图G的邻接矩阵并输出" << endl;
    VertexType Vex[MaxVertex] = { 0,1,2,3,4,5 };                    // 顶点表
    MGraph MG(Vex, 6, 10);                                          // 建立G的邻接矩阵
    MG.PrintMGraph();                                               // 输出G的邻接矩阵
    cout << endl << "（2）建立如图所示的有向图G的邻接表并输出" << endl;
    ALGraph ALG(Vex, 6, 10);
    ALG.PrintALG();
    return 0;
}
```

第 2 题：

// travsal.h

```
// 为精简报告长度，栈、队列具体实现不添加进报告（因为非主要内容）
// 在程序源代码文件中提供栈、队列具体实现

#pragma once
#include <iostream>
#include "linkstack.h"
#include "sqqueue.h"
using namespace std;

#define MaxVertex 10              // 顶点数目的最大值
#define INFINITY 2147483647       // 无穷
typedef int VertexType;         // 顶点的数据类型
typedef int EdgeType;             // 带权图边权值的数据类型

// 为了便捷使用，定义一个结构体来输入图
struct GraphInfo
{
    int Vex1;     // 起点
    int Vex2;     // 终点
    int Weight;   // 权值
};

// ------------- 邻接矩阵表示法 -------------
```

```
class MGraph
{
public:
    MGraph(VertexType* V, const GraphInfo* GI, int n, int e);
    void DFS(int v);
    void DFSTraverse();
    void DFSTraverse_NoRa(int v);
    void BFSTraverse(int v);

private:
    VertexType Vex[MaxVertex];                  // 顶点表
    EdgeType Edge[MaxVertex][MaxVertex];    // 邻接矩阵，边表
    bool Visited[MaxVertex];                    // 访问标记数组
    int VexNum;                                 // 当前顶点数
    int ArcNum;                                 // 当前弧数
};

// ---------- 邻接表表示法 ----------
// 边表结点
typedef struct ArcNode
{
    int adjvex;             // 邻接点域
    EdgeType weight;        // 边的权重
    struct ArcNode* next;   // 指针域
}ArcNode;
// 顶点表结点
typedef struct VertxNode
{
    VertexType data;        // 顶点域
    ArcNode* first;         // 边表的头指针
}VNode, AdjList[MaxVertex];
// 邻接表储存的图
class ALGraph
{
public:
    ALGraph(VertexType* Vex, const  GraphInfo* GI, int n, int e);
    ~ALGraph();
    void DFS(int v);
    void DFSTraverse();
    void DFSTraverse_NoRa(int v);
    void BFSTraverse(int v);

private:
    AdjList vertices;           // 邻接表
```

```
    bool Visited[MaxVertex];  // 访问标记数组
    int VexNum;               // 当前顶点数
    int ArcNum;               // 当前弧数
};
```

// travsal.cpp

```
////////////////////////////////////////////////////////////////////////////
//////////////////////////////以下为图实现/////////////////////////////////
#include "travsal.h"
#include <iostream>
using namespace std;

// -------------- 邻接矩阵表示法 --------------
// 图邻接矩阵的构造函数
MGraph::MGraph(VertexType* V, const GraphInfo* GI, int n, int e)
{
    // 根据V[]初始化顶点信息与访问标记数组
    for (int j = 0; j < n; j++)
    {
        this->Vex[j] = j;
        this->Visited[j] = false;
    }
    // 初始化图的顶点数和弧数
    this->VexNum = n;
    this->ArcNum = e;
    // 初始化图邻接矩阵
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            this->Edge[i][k] = INFINITY;
    // 根据输入图的边，构造邻接矩阵
    for (int j = 0; j < e; j++)
        this->Edge[GI[j].Vex1][GI[j].Vex2] = GI[j].Weight;

}
// 连通图（连通分量）的邻接矩阵的深度优先遍历    [递归]
void MGraph::DFS(int v)
{
    // 访问该顶点
    cout << Vex[v] << " ";
    // 标记为已访问
    Visited[v] = true;
    // 查看是否有邻接顶点没有访问的
    for (int i = 0; i < VexNum; i++)
    {
```

```
            // 如果存在邻接顶点i
            if (Edge[v][i] != INFINITY)
            {
                // 查看访问标记
                // 如果没有被访问过
                if (!Visited[i])
                    DFS(i);
            }
        }
}
// 图的邻接矩阵的深度优先遍历(非连通图也可以)[递归]
void MGraph::DFSTraverse()
{
    // 重置访问标记
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    // 深度优先遍历
    // 当访问标记数组存在false时一直调用DFS()
    // 这样可以保证每个图的每个连通分量都可以遍历到
    // 这里是从顶点0开始深度优先遍历
    for (int i = 0; i < VexNum; i++)
        if (!Visited[i])
            DFS(i);
}
// 图的邻接矩阵的深度优先遍历[非递归]
void MGraph::DFSTraverse_NoRa(int v)
{
    // 初始化辅助栈
    LinkStack S;
    InitStack(S);
    // 重置访问标记
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    Push(S, v);
    Visited[v] = true;
    while (!StackEmpty(S))
    {
        // 出栈
        Pop(S, v);
        // 访问
        cout << Vex[v] << " ";
        // 寻找邻接顶点
        for (int i = 0; i < VexNum; i++)
        {
```

```
            // 如果是邻接顶点
            if (Edge[v][i] != INFINITY)
            {
                // 看看有没有被访问
                if (!Visited[i])
                {
                    Push(S, i);
                    Visited[i] = true;
                }
            }
        }
    }
    DestoryStack(S);
}
// 图的邻接矩阵的广度优先遍历
void MGraph::BFSTraverse(int v)
{
    // 初始化辅助队列
    SqQueue Q;
    InitQueue(Q);
    // 初始化访问标记数组
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    // 访问
    cout << Vex[v] << " ";
    Visited[v] = true;
    // 入队
    EnQueue(Q, v);
    // 当队列非空时
    while (!QueueEmpty(Q))
    {
        // 队头出队
        DeQueue(Q, v);
        // 找队头的第一个邻接点
        for (int i = 0; i < VexNum; i++)
        {
            // !=INFINITY说明有边
            if (Edge[v][i] != INFINITY)
            {
                // 没被访问过就访问一下
                if (!Visited[i])
                {
                    cout << Vex[i] << " ";
                    Visited[i] = true;
```

```
                    // 入队
                    // 等该"层"都入队了
                    // 再通过队列访问下一层
                    EnQueue(Q, i);
                }
            }
        }
    }
    DestroyQueue(Q);
}


// ---------- 邻接表表示法 ----------
// 图邻接表的构造函数
ALGraph::ALGraph(VertexType* Vex, const GraphInfo* GI, int n, int e)
{
    this->ArcNum = e;
    this->VexNum = n;
    // 初始化邻接表与访问标记数组
    for (int i = 0; i < n; i++)
    {
        this->vertices[i].data = Vex[i];
        this->vertices[i].first = NULL;
        this->Visited[i] = false;
    }
    // 根据输入图的边，构造邻接表
    for (int j = 0; j < e; j++)
    {
        ArcNode* s = new ArcNode;
        s->adjvex = GI[j].Vex2;
        s->weight = GI[j].Weight;
        // 前插法
        s->next = vertices[GI[j].Vex1].first;
        vertices[GI[j].Vex1].first = s;
    }
}
// 图邻接表的析构函数
ALGraph::~ALGraph()
{
    ArcNode* q = NULL;  // 前驱工作指针
    ArcNode* p = NULL;  // 后驱工作指针
    for (int i = 0; i < VexNum; i++)
    {
        p = q = vertices[i].first;
```

```
        while (p)
        {
            q = p;
            p = p->next;
            delete q;
        }
        vertices[i].data = -1;
    }
    VexNum = 0;
    ArcNum = 0;
    cout << endl << endl << "邻接表销毁已完成！" << endl;
}
// 连通图（连通分量）的邻接表的深度优先遍历
void ALGraph::DFS(int v)
{
    ArcNode* p = NULL;
    // 访问该顶点
    cout << vertices[v].data << " ";
    // 标记为已访问
    Visited[v] = true;
    // 查看邻接顶点
    p = vertices[v].first;
    while (p)
    {
        // 如果存在邻接顶点p->adjvex
        // 查看访问标记
        // 如果没有被访问过,就递归访问
        if (!Visited[p->adjvex])
            DFS(p->adjvex);
        p = p->next;
    }
}
// 图的邻接表的深度优先遍历（非连通图也可以）
void ALGraph::DFSTraverse()
{
    // 重置访问标记
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    // 深度优先遍历
    // 当访问标记数组存在false时一直调用DFS()
    // 这样可以保证每个图的每个连通分量都可以遍历到
    // 这里是从顶点0开始深度优先遍历
    for (int i = 0; i < VexNum; i++)
        if (!Visited[i])
```

```
            DFS(i);
}
// 图的邻接表的深度优先遍历[非递归]
void ALGraph::DFSTraverse_NoRa(int v)
{
    // 工作指针
    ArcNode* p = NULL;
    // 初始化辅助栈
    LinkStack S;
    InitStack(S);
    // 重置访问标记
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    Push(S, v);
    Visited[v] = true;
    while (!StackEmpty(S))
    {
        // 出栈
        Pop(S, v);
        // 访问
        cout << vertices[v].data << " ";
        // 寻找邻接顶点
        p = vertices[v].first;
        while (p)
        {
            // 看看有没有被访问
            if (!Visited[p->adjvex])
            {
                Push(S, p->adjvex);
                Visited[p->adjvex] = true;
            }
            p = p->next;
        }
    }
    DestoryStack(S);
}
// 图的邻接表的广度优先遍历
void ALGraph::BFSTraverse(int v)
{
    // 工作指针
    ArcNode* p = NULL;
    // 初始化辅助队列
    SqQueue Q;
    InitQueue(Q);
```

```
    // 初始化访问标记数组
    for (int i = 0; i < VexNum; i++)
        Visited[i] = false;
    // 访问
    cout << vertices[v].data << " ";
    Visited[v] = true;
    // 入队
    EnQueue(Q, v);
    // 当队列非空时
    while (!QueueEmpty(Q))
    {
        // 队头出队
        DeQueue(Q, v);
        // 找队头的第一个邻接点
        p = vertices[v].first;
        while (p)
        {
            // 如果没有访问过
            if (!Visited[p->adjvex])
            {
                // 访问
                cout << vertices[p->adjvex].data << " ";
                Visited[p->adjvex] = true;
                // 入队
                EnQueue(Q, p->adjvex);
            }
            p = p->next;
        }
    }
    DestroyQueue(Q);
}
```

// exp8-2.cpp

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

int main()
{
    const GraphInfo Graph8_54[10] =
{ {0,1,5},{1,2,4},{3,2,5},{4,3,5},{5,4,1},{5,0,3},{2,0,8},{0,3,7},{3,5,6},{2,5,9} };
    VertexType Vex[MaxVertex] = { 0,1,2,3,4,5 };       // 顶点表
    cout << "（0）建立图，本程序实现邻接矩阵与邻接表的深度、广度优先遍历算法" << endl;
    MGraph MG(Vex, Graph8_54, 6, 10);                  // 建立邻接矩阵图MG
```

```
    ALGraph ALG(Vex, Graph8_54, 6, 10);                // 建立邻接表图ALG
    cout << "（1）输出如图的有向图G从顶点0开始的深度优先遍历序列（递归算法）" << endl;
    cout << "邻接矩阵：";
    MG.DFSTraverse();
    cout << endl << "邻接表：";
    ALG.DFSTraverse();
    cout << endl;
    cout << endl << "（2）输出如图的有向图G从顶点0开始的深度优先遍历序列（非递归算法）" << endl;
    cout << "邻接矩阵：";
    MG.DFSTraverse_NoRa(0);
    cout << endl << "邻接表：";
    ALG.DFSTraverse_NoRa(0);
    cout << endl;
    cout << endl << "（3）输出如图的有向图G从顶点0开始的广度优先遍历序列" << endl;
    cout << "邻接矩阵：";
    MG.BFSTraverse(0);
    cout << endl << "邻接表：";
    ALG.BFSTraverse(0);
    return 0;
}
```

第 3 题：

//exp8-5.cpp

```
#include <iostream>
using namespace std;

#define MaxVertex 10              // 顶点数目的最大值
#define INFINITY 2147483647       // 无穷
typedef int VertexType;         // 顶点的数据类型
typedef int EdgeType;             // 带权图边权值的数据类型

//////////////////////////以下为结构体定义/////////////////////////

// 为了便捷使用，定义一个结构体来输入图
struct GraphInfo
{
    int Vex1;         // 起点
    int Vex2;         // 终点
    int Weight;       // 权值
};
// 最小生成树的辅助数组结构
```

```
struct ShortEdge
{
    int lowcost;  // 边的权值
    int adjvex;   // 邻接顶点
};
//  邻接矩阵表示法
class MGraph
{
public:
    MGraph(VertexType* V, const GraphInfo* GI, int n, int e);
    void PrintMGraph();
    int MiniEdge(ShortEdge* SE);
    void OutputSMT(ShortEdge* SE, int k);
    void Prim(int start);

private:
    VertexType Vex[MaxVertex];                  // 顶点表
    EdgeType Edge[MaxVertex][MaxVertex];    // 邻接矩阵，边表
    int VexNum;                                 // 当前顶点数
    int ArcNum;                                 // 当前弧数
};


//////////////////////以下为函数实现//////////////////////

// 构造
MGraph::MGraph(VertexType* V, const GraphInfo* GI, int n, int e)
{
    // 根据V[]初始化顶点信息与访问标记数组
    for (int j = 0; j < n; j++)
        this->Vex[j] = j;
    // 初始化图的顶点数和弧数
    this->VexNum = n;
    this->ArcNum = e;
    // 初始化图邻接矩阵
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            this->Edge[i][k] = INFINITY;
    // 根据输入图的边，构造无向图邻接矩阵
    for (int j = 0; j < e; j++)
    {
        this->Edge[GI[j].Vex1][GI[j].Vex2] = GI[j].Weight;
        this->Edge[GI[j].Vex2][GI[j].Vex1] = GI[j].Weight;
    }
```

```
}
// 打印图邻接矩阵
void MGraph::PrintMGraph()
{
    cout << endl << "输出图的邻接矩阵如下：" << endl;
    for (int i = 0; i < this->VexNum; i++)
    {
        for (int k = 0; k < this->VexNum; k++)
        {
            if (this->Edge[i][k] == INFINITY)
                cout << "∞\t";
            else
                cout << this->Edge[i][k] << "\t";
        }
        cout << endl;
    }
}
// 在辅助数组内寻找最小权值的边
int MGraph::MiniEdge(ShortEdge* SE)
{
    int min = -1;
    int mincost = INFINITY;
    for (int i = 0; i < this->VexNum; i++)
    {
        // 节点没有加入U且有边
        if (SE[i].lowcost != 0 && SE[i].lowcost != INFINITY)
        {
            if (mincost > SE[i].lowcost)
            {
                // 更新最小值
                mincost = SE[i].lowcost;
                // 记录下标
                min = i;
            }
        }
    }
    return min;
}
// 输出上函数中找到的最小边
void MGraph::OutputSMT(ShortEdge* SE, int k)
{
    cout << "(" << SE[k].adjvex << "," << k << ") " << SE[k].lowcost << endl;
}
```

```
// Prim算法求最小生成树
void MGraph::Prim(int start)
{
    int k = -1;
    // 初始化辅助数组SE
    ShortEdge* SE = new ShortEdge[VexNum];
    for (int i = 0; i < VexNum; i++)
    {
        SE[i].adjvex = start;
        SE[i].lowcost = Edge[start][i];
    }
    // 起点srtart加入集合U
    // ！！！加入集合U标志为lowcost=0
    SE[start].lowcost = 0;
    for (int i = 0; i < VexNum - 1; i++)
    {
        // 在U与V-U之间查找权值最小的边，k为边所连接顶点
        k = MiniEdge(SE);
        // 打印输出这一条边
        OutputSMT(SE, k);
        // 把顶点加入集合U
        SE[k].lowcost = 0;
        // 接下来在在U与V-U之间查找权值最小的边
        // 需要同时考虑在U里的所有结点与U-V结点的关系
        // 假设现在lowcost初始化后无改动
        // 将现在的SE数组里的lowcost与新加入节点k所有邻接边权值比较
        // 将较小的权值替换到SE数组的lowcost中去
        // 这样下一次循环只要MiniEdge(SE)就可以找到U内最小权值的新边k
        // 每次循环都这样做，将k与lowcost比较替换
        for (int j = 0; j < VexNum; j++)
        {
            if (SE[j].lowcost > Edge[k][j])
            {
                // 更改邻接顶点域
                // 表示是集合U内顶点到k这个顶点的最小权值
                SE[j].adjvex = k;
                // 将较小的权值替换到SE数组的lowcost
                SE[j].lowcost = Edge[k][j];
            }
        }
    }
}
```

```
//////////////////////////main//////////////////////////////
int main()
{
    const GraphInfo Graph8_55[10] =
{ {0,1,5},{1,2,4},{3,2,5},{4,3,5},{5,4,1},{5,0,3},{2,0,8},{0,3,7},{3,5,6},{2,5,9} };
    VertexType Vex[MaxVertex] = { 0,1,2,3,4,5 };      // 顶点表
    cout << "采用Prim算法求图8.5.5从顶点0开始的最小生成树" << endl;
    MGraph MG(Vex, Graph8_55, 6, 10);                 // 建立邻接矩阵图MG
    MG.PrintMGraph();
    cout << endl << "最小生成树：" << endl;
    MG.Prim(0);
    return 0;

}
```

第 4 题：

```
// exp8-10.cpp
#include <iostream>
using namespace std;

#define MaxVertex 10            // 顶点数目的最大值
#define INFINITY 21474          // 无穷
typedef int VertexType;         // 顶点的数据类型
typedef int EdgeType;           // 带权图边权值的数据类型

//////////////////////////以下为结构体定义//////////////////////

// 为了便捷使用，定义一个结构体来输入图
struct GraphInfo
{
    int Vex1;         // 起点
    int Vex2;         // 终点
    int Weight = 1;   // 权值
};
// 最小生成树的辅助数组结构
struct ShortEdge
{
    int lowcost;  // 边的权值
    int adjvex;   // 邻接顶点
};
```

```
//  邻接矩阵表示法
class MGraph
{
public:
    MGraph(VertexType* V, const GraphInfo* GI, int n, int e);
    void PrintMGraph();
    void EmptyVisited();
    void FindPath(int start, int end, int depth = -1);
    void FindLenPath(int start, int end, int len, int depth = -1);
    void Floyd(int start, int end);

private:
    VertexType Vex[MaxVertex];                  // 顶点表
    EdgeType Edge[MaxVertex][MaxVertex];    // 邻接矩阵，边表
    bool Visited[MaxVertex];                    // 访问标记数组
    int VexNum;                                 // 当前顶点数
    int ArcNum;                                 // 当前弧数
};


//////////////////////////以下为函数实现//////////////////////////

// 构造
MGraph::MGraph(VertexType* V, const GraphInfo* GI, int n, int e)
{
    // 根据V[]初始化顶点信息与访问标记数组
    for (int j = 0; j < n; j++)
    {
        this->Vex[j] = j;
        this->Visited[j] = false;
    }
    // 初始化图的顶点数和弧数
    this->VexNum = n;
    this->ArcNum = e;
    // 初始化图邻接矩阵
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            this->Edge[i][k] = INFINITY;
    // 根据输入图的边，构造无向图邻接矩阵
    for (int j = 0; j < e; j++)
        this->Edge[GI[j].Vex1][GI[j].Vex2] = GI[j].Weight;
}
void MGraph::PrintMGraph()
{
```

```
    for (int i = 0; i < this->VexNum; i++)
    {
        for (int k = 0; k < this->VexNum; k++)
        {
            if (this->Edge[i][k] == INFINITY)
                cout << "∞\t";
            else
                cout << this->Edge[i][k] << "\t";
        }
        cout << endl;
    }
    cout << endl;
}
void MGraph::EmptyVisited()
{
    for (int j = 0; j < VexNum; j++)
        this->Visited[j] = false;
}
void MGraph::FindPath(int start, int end, int depth)
{
    static int path[MaxVertex] = {};
    depth++;
    // 访问start
    path[depth] = start;
    Visited[start] = true;
    // 采用深度优先遍历，递归
    // 递归终止条件
    if (start == end)
    {
        // 输出路径
        for (int i = 0; i < depth + 1; i++)
        {
            cout << path[i];
            if (i != depth)
                cout << " -> ";
            else
                cout << endl;
        }
        Visited[start] = false;
        return;
    }
    // 查看start有没有邻接顶点
    for (int i = 0; i < VexNum; i++)
    {
```

```
        if (Edge[start][i] != INFINITY)
        {
            if (!Visited[i])
                FindPath(i, end, depth);
        }
    }
    // 回溯的时候消除start访问标记
    Visited[start] = false;
}
void MGraph::FindLenPath(int start, int end, int len, int depth)
{
    static int path[MaxVertex] = {};
    depth++;
    // 访问start
    path[depth] = start;
    Visited[start] = true;
    // 采用深度优先遍历，递归
    // 递归终止条件，递归深度=长度-1
    if (depth > len - 1)
    {
        Visited[start] = false;
        return;
    }
    // start=end找到了
    if (start == end)
    {
        // 输出路径
        for (int i = 0; i < depth + 1; i++)
        {
            cout << path[i];
            if (i != depth)
                cout << " -> ";
            else
                cout << endl;
        }
        // 找到了
        // 回溯的时候消除start访问标记
        Visited[start] = false;
        return;
    }
    // 查看start有没有邻接顶点
    for (int i = 0; i < VexNum; i++)
    {
        if (Edge[start][i] != INFINITY)
```

```
        {
            if (!Visited[i])
                FindLenPath(i, end, len, depth);
        }
    }
    // 这里是找完这个顶点的所有路径了，要回溯了
    // 回溯的时候消除start访问标记
    Visited[start] = false;
}
void MGraph::Floyd(int start, int end)
{
    // 存储两个点之间最短路径的长度信息
    VertexType A[MaxVertex][MaxVertex];
    // 存储两个点之间最短路径的信息（经谁中转）
    VertexType path[MaxVertex][MaxVertex];
    // 初始化path和数组A
    for (int i = 0; i < VexNum; i++)
    {
        for (int k = 0; k < VexNum; k++)
        {
            A[i][k] = Edge[i][k];
            path[i][k] = -1;
        }
    }
    // 外层循环表示添加一个顶点Vi作为中转
    for (int i = 0; i < VexNum; i++)
    {
        // 内层双循环用来遍历整个矩阵
        for (int j = 0; j < VexNum; j++)
        {
            for (int k = 0; k < VexNum; k++)
            {
                // 当A里的值已经不是最短路径时
                // 更新为通过中转点Vi的路径长
                // ！！！这里体现的是迭代的思想
                // A里的值永远保持最优（这里和最小生成树的算法非常像）
                if (A[j][k] > A[j][i] + A[i][k])
                {
                    // 更新A[][]~
                    A[j][k] = A[j][i] + A[i][k];
                    // 在path里说明是通过谁中转的
                    path[j][k] = i;
                }
            }
```

```
        }
    }
    cout << start << " -> ";
    if (path[start][end] != -1)
        cout << path[start][end] << " -> ";
    cout << end << endl;
}

//////////////////////////////main//////////////////////////////
int main()
{
    const GraphInfo Graph8_56[12] =
{ {0,1},{1,2},{3,2},{4,3},{5,4},{5,0},{2,0},{0,3},{3,5},{2,5},{5,1},{5,3} };
    VertexType Vex[MaxVertex] = { 0,1,2,3,4,5 };        // 顶点表
    MGraph G(Vex, Graph8_56, 6, 12);                    // 建立邻接矩阵图G
    cout << "（0）输出图8.56的邻接矩阵" << endl;
    G.PrintMGraph();
    cout << "（1）输出如图8.56的有向图G从顶点5到顶点2的所有简单路径" << endl;
    G.FindPath(5, 2);
    cout << endl << "（2）输出如图8.56的有向图G从顶点5到顶点2的所有长度为3的简单路径" << endl;
    G.FindLenPath(5, 2, 3);
    cout << endl << "（3）输出如图8.56的有向图G从顶点5到顶点2的最短路径" << endl;
    G.Floyd(5, 2);
    return 0;

}
```

第 5 题：

```
// exp8-14
#include <iostream>
using namespace std;

#define M 6                                 // 迷宫矩阵行数
#define N 6                                 // 迷宫矩阵列数
#define MaxSize (M-2)*(N-2)       // 顶点数目的最大值

bool Visited[MaxSize] = {};             // 访问标记数组

// 点坐标表示
struct Point
{
```

```
    int row = -1;          // 行号
    int column = -1; // 列号
};
// 边表结点
typedef struct ArcNode
{
    int adjvex;                    // 邻接点坐标域
    struct ArcNode* next;     // 指针域
}ArcNode;
// 顶点表结点
typedef struct VertxNode
{
    Point data;            // 顶点坐标域
    ArcNode* first;        // 边表的头指针
}VNode, AdjList[MaxSize];
// 邻接表储存的图
class ALGraph
{
public:
    ALGraph(const int mz[M][N]);
    ~ALGraph();
    void PrintALG();
    bool IsNeighbour(const int mz[M][N], Point& p, int direct);
    int FindPoint(const int mz[M][N], Point p);
    void AddArc(int count, int arc);
    void FindPath(int start, int end, int depth = -1);

private:
    AdjList vertices;          // 邻接表
    int VexNum;                // 当前顶点数
};

// 由迷宫矩阵构造图的邻接表
ALGraph::ALGraph(const int mz[M][N])
{
    Point temp;
    // 邻接表的下标
    int count = 0;
    // 初始化邻接表
    // 实际上就是在给可以走的点编号
    for (int i = 0; i < MaxSize; i++)
        vertices[i].first = NULL;

    for (int i = 1; i < M - 1; i++)
```

```
        for (int j = 1; j < N - 1; j++)
            if (mz[i][j] == 0)
            {
                // 邻接表顶点信息记录迷宫矩阵坐标
                vertices[count].data.row = i;
                vertices[count++].data.column = j;
            }

    VexNum = count;
    count = 0;
    // 遍历迷宫矩阵生成
    // i行j列进行矩阵遍历
    for (int i = 1; i < M - 1; i++)
    {
        for (int j = 1; j < N - 1; j++)
        {
            // 如果路可以走
            if (mz[i][j]==0)
            {
                // 开始建立边表
                // 在一个迷宫中一个顶点最多有4条边，分别是上下左右
                // 搜索上下左右
                for (int k = 0; k < 4; k++)
                {
                    temp.row = i;
                    temp.column = j;
                    // 可达就添加边表
                    if (IsNeighbour(mz, temp, k))
                        AddArc(count, FindPoint(mz, temp));
                }
                count++;
            }
        }
    }
}
// 图邻接表的析构函数
ALGraph::~ALGraph()
{
    ArcNode* q = NULL; // 前驱工作指针
    ArcNode* p = NULL; // 后驱工作指针
    for (int i = 0; i < VexNum; i++)
    {
        Point temp = { -1,-1 };
        p = q = vertices[i].first;
```

```
        while (p)
        {
            q = p;
            p = p->next;
            delete q;
        }
        vertices[i].data = temp;
    }
    VexNum = 0;
}
// 打印邻接表
void ALGraph::PrintALG()
{
    ArcNode* p = NULL;      // 工作指针
    cout << "输出图的邻接表如下：" << endl;
    for (int i = 0; i < VexNum; i++)
    {
        cout << "[" << i << "] ";
        cout << "(" << vertices[i].data.row << " ,";
        cout << vertices[i].data.column << "): ";
        p = vertices[i].first;
        while (p)
        {
            cout << "(" << vertices[p->adjvex].data.row << ", ";
            cout << vertices[p->adjvex].data.column << ") ";
            p = p->next;
        }
        cout << endl;
    }
}
// 根据迷宫坐标查找邻接表的下标
int ALGraph::FindPoint(const int mz[M][N], Point p)
{
    for (int j = 0; j < MaxSize; j++)
        if (vertices[j].data.row == p.row && vertices[j].data.column == p.column)
            return j;
}
// 在编号为count的邻接表中添加一条到arc的边
void ALGraph::AddArc(int count, int arc)
{
    ArcNode* s = new ArcNode;
    s->adjvex = arc;
    s->next = vertices[count].first;
    vertices[count].first = s;
```

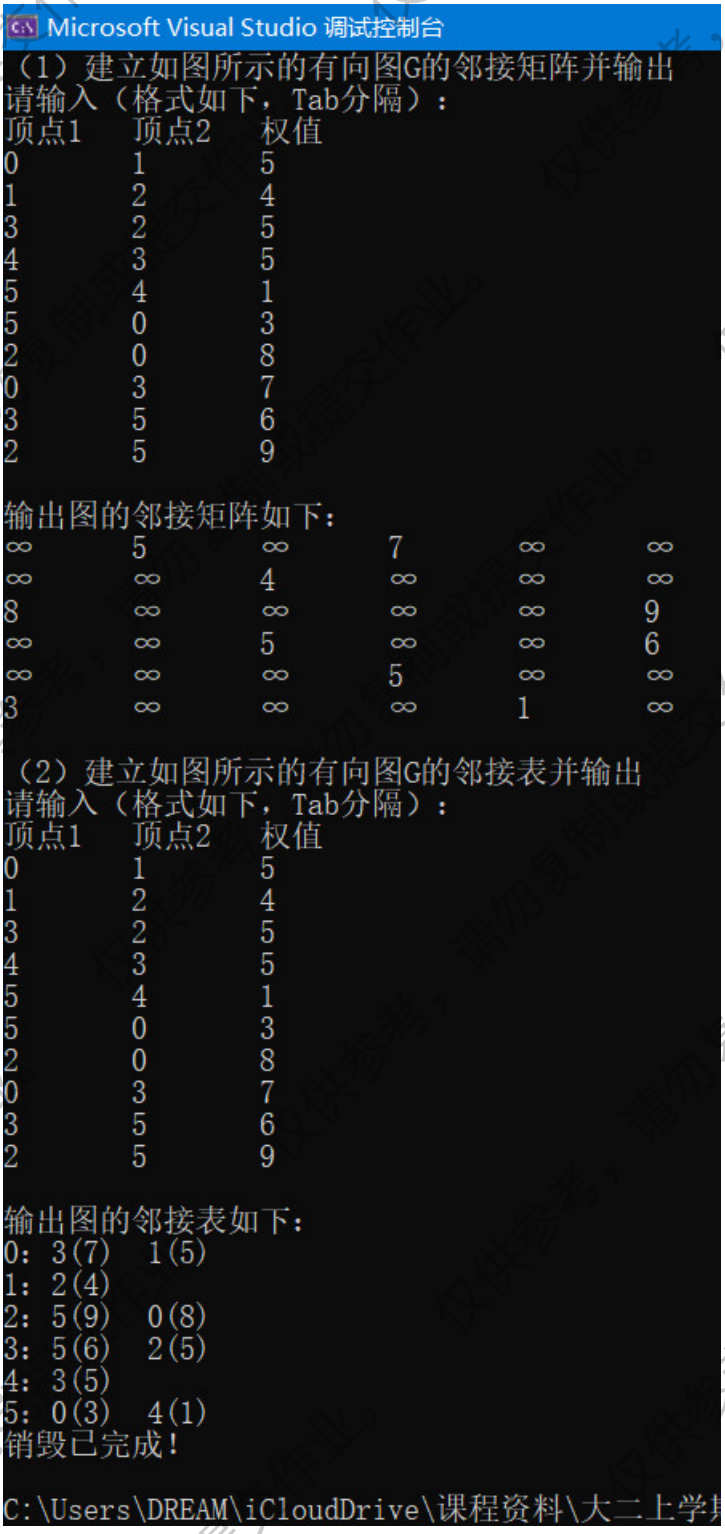
```
}
// 查找在坐标p的上下左右是否可达
// direct：0右，1下，2左，3上
bool ALGraph::IsNeighbour(const int mz[M][N], Point& p, int direct)
{
    switch (direct)
    {
    case 0:// 右
        p.column += 1;
        break;
    case 1:// 下
        p.row += 1;
        break;
    case 2:// 左
        p.column -= 1;
        break;
    case 3:// 上
        p.row -= 1;
        break;
    }
    if (!mz[p.row][p.column])
        return true;
    else
        return false;
}
// 连通图（连通分量）的邻接表的深度优先遍历
void ALGraph::FindPath(int start, int end, int depth)
{
    ArcNode* p = NULL;
    static Point path[MaxSize] = {};
    depth++;
    // 访问start
    path[depth] = vertices[start].data;
    Visited[start] = true;
    // 采用深度优先遍历，递归
    // 递归终止条件
    if (start == end)
    {
        // 输出路径
        for (int i = 0; i < depth + 1; i++)
        {
            cout << "(" << path[i].row << ", ";
            cout << path[i].column << ")";
            if (i != depth)
```

```
                cout << "->";
            else
                cout << endl;
        }
        Visited[start] = false;
        return;
    }
    // 查看start有没有邻接顶点
    p = vertices[start].first;
    while (p)
    {
        if (!Visited[p->adjvex])
            FindPath(p->adjvex, end, depth);
        p = p->next;
    }
    // 回溯的时候消除start访问标记
    Visited[start] = false;
}

int main()
{
    // 迷宫矩阵
    // 1表示障碍物，0反之
    Point startPoint = { 1,1 };
    Point endPoint = { M - 2,N - 2 };
    const int maze[M][N] = {
        { 1,1,1,1,1,1 },
        { 1,0,0,0,1,1 },
        { 1,0,1,0,0,1 },
        { 1,0,0,0,1,1 },
        { 1,1,0,0,0,1 },
        { 1,1,1,1,1,1 } };

    ALGraph MZGraph(maze);
    cout << "（1）建立一个迷宫对应的邻接表表示" << endl;
    MZGraph.PrintALG();
    int start = MZGraph.FindPoint(maze, startPoint);
    int end = MZGraph.FindPoint(maze, endPoint);
    cout << "\n（2）采用深度优先遍历算法输出从入口（1，1）到出口（M,N）的所有迷宫路径。\n";
    MZGraph.FindPath(start, end);
    return 0;

}
```

五、实验结果及分析

1、

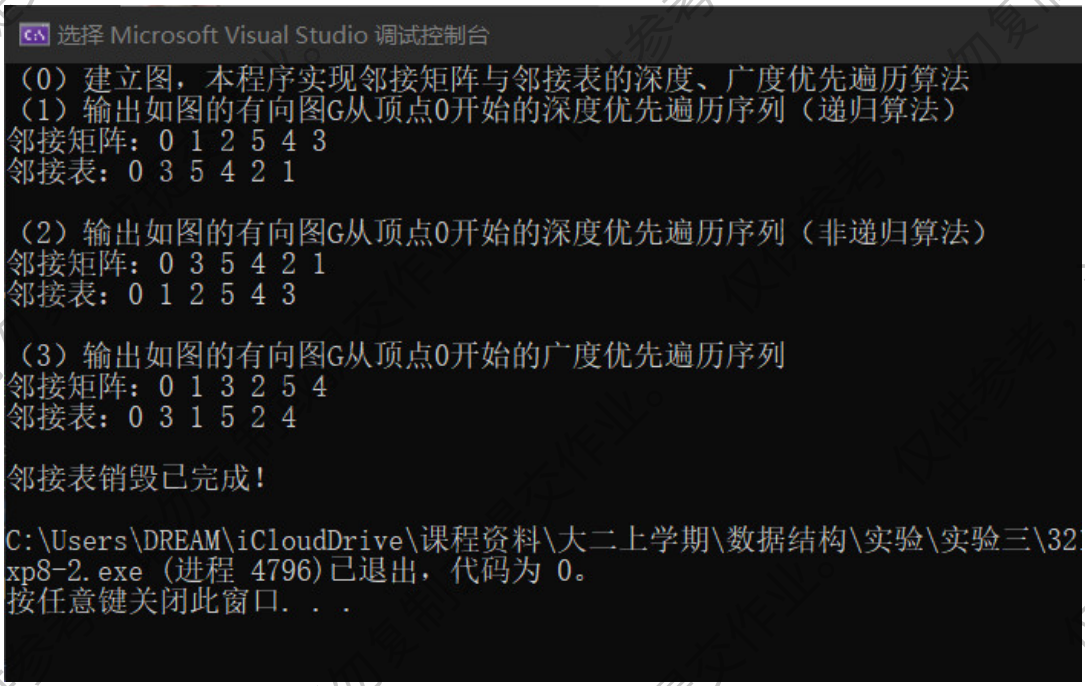
运行结果：



运行结果正确

2、

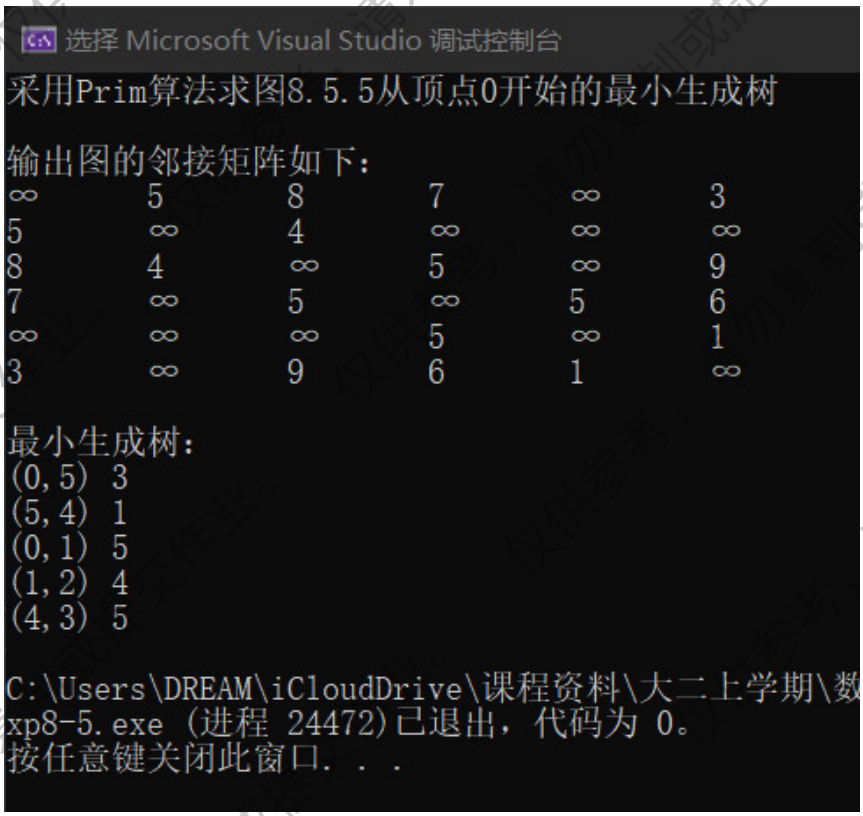
运行结果：



运行结果正确

3、

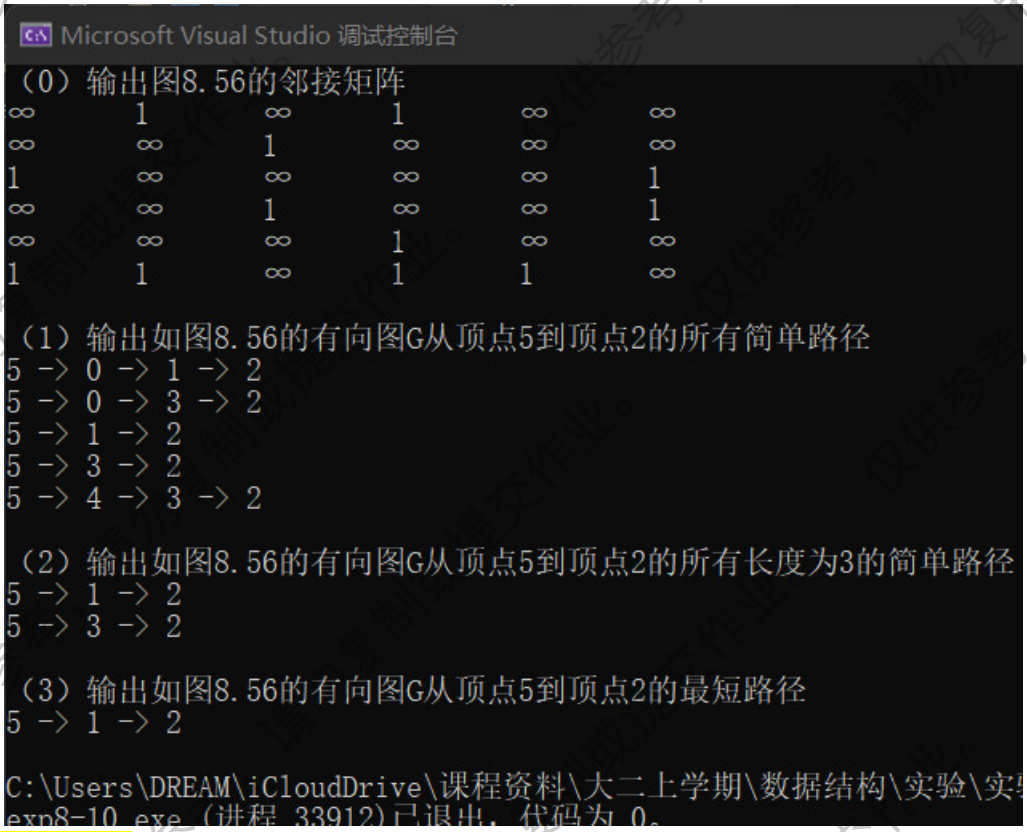
运行结果：



运行结果正确

4、

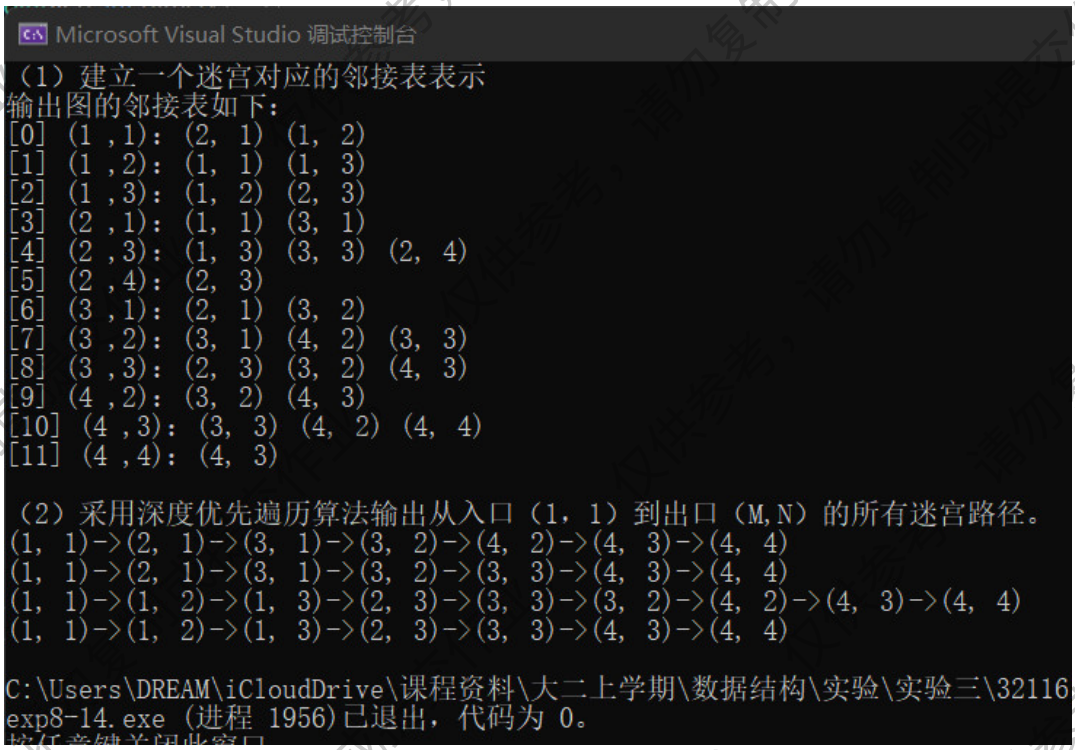
运行结果：



运行结果正确

5、

运行结果：



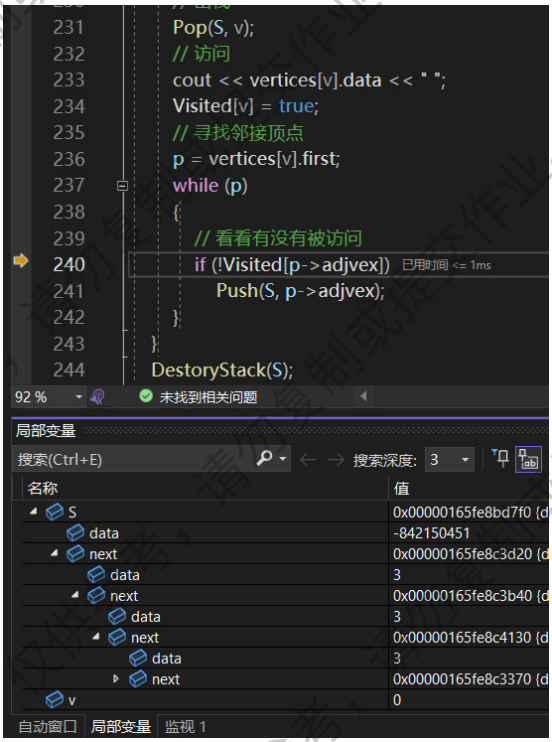
运行结果正确

调试&优化日志：

1) **优化：**在第一题中我采取的是手动输入每一条边的形式建立图，因为这种方法不利于调试，后续题目中我设计了一个结构体来代替手动输入每一条边。

```
struct GraphInfo
{
    int Vex1;       // 起点
    int Vex2;       // 终点
    int Weight;     // 权值
};
```

2) **bug：**在写第二题时的深度优先遍历非递归算法时，我没有在进栈后立即将标记数组设置为true，导致了死循环产生



3) **bug：**在第四题的最短路径中，我的图使用 `INFINITY 2147483647` 来表示两个顶点之间没有边，但是 `floyd` 算法需要将辅助数组 `A` 里的值与添加中转点 `Vi` 后的权值比较。但是如果恰好为两个没有边的顶点相加时会导致 `int` 溢出，导致 `bug`

解决方法为将 `INFINITY` 值改小

```
// 当A里的值已经不是最短路径时
// 更新为通过中转点Vi的路径长
// ！ ！ ！ 这里体现的是迭代的思想
// A里的值永远保持最优（这里和最小生成树的算法非常
if (A[j][k] > Edge[j][i] + Edge[i][k])
```

实验体会与心得：

这次的实验为图的相关操作实验。图的操作函数不像树，在树的相关操作中基本上都涉及到递归的思想和使用，只有在使用深度优先遍历的时候才有可能会使用到的递归函数。并且在使用深度优先算法寻路中也同时使用了回溯的相关思想，递归返回时恢复原来的样子，消除“痕迹”。

本次实验给我的这道感觉是图的相关操作时间复杂度都比较高，如图的最短路径算法，需要嵌套三层的 for 循环。同时算法的思想也更抽象，需要有比较好的抽象思维能力。