实验六 图实验一

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 | 张渊 | 学号 | 2022210000 | 专业班级 | 计科22-0 |
| 指导教师 | 张先宜 | 实验时间 | 2023.6.11 | 实验地点 |  |

6.1 **实验目的**

(1) 掌握图的基本概念。

(2) 掌握图的存储结构的设计与实现，基本运算的实现。

(3) 熟练掌握图的两种遍历算法、遍历生成树及遍历算法的应用。

(4 掌握图的Prim和Kruskal最小生成树算法。

6.2 **实验要求**

1. 结构定义和算法实现放入库文件，如“graph.h”中；
2. 图的测试数据用文本文件方式给出，例如测试数据名为dn.grp的有向网，可参考发来的图形状和参考存储文件；
3. 图创建方法可自行选择；
4. 可多次连续测试。

6.3 **实验任务**

分别设计图（网）的邻接矩阵、邻接表存储结构，编写算法实现下列问题的求解。

1．打印出图（网）的两种遍历序。

实验测试数据基本要求：

第一组数据： udg8.grp

第二组数据： udg115.grp

第三组数据： dg6.grp

第四组数据： f14.grp

2．求给定图中的边（或弧）的数目。 实验测试数据基本要求：第一组数据： udg8.grp

第二组数据： udg115.grp

第三组数据： dg6.grp

第四组数据： f14.grp

3．对给定的图G及出发点v0，设计算法从V0出发深度优先遍历图G，并构造出相应的生成树或生成森林。

实验测试数据基本要求：

第一组数据： udg8.grp

第二组数据： dg6.grp

第三组数据： udn8.grp

第四组数据： dn10.grp

4．对给定的图G及出发点v0，设计算法从V0出发广度优先遍历图G，并构造出相应的生成树或生成森林。

第一组数据： udg8.grp

第二组数据： dg6.grp

第三组数据： un8.grp

第四组数据： dn10.grp

5．实现Prim算法，求解下列给定图G的最小生成树。 实验测试数据基本要求：第一组数据： udn6.grp

第二组数据： un8.grp

6．实现Kruskal算法，求解下列给定图G的最小生成树。 实验测试数据基本要求：第一组数据： udn6.grp

第二组数据： un8.grp

**6.4数据结构设计**

邻接矩阵

typedef char elementType; //定义图中顶点的数据类型

typedef int cellType; //定义邻接矩阵中元素的数据类型

//对无权图，1-相邻（有边），0-不相邻（无边）

//对有权图，为边的权值，特别是无穷大。

typedef enum { UDG, UDN, DG, DN } GraphKind; //枚举图的类型--无向图，无向网，有向图，有向网

typedef struct GraphAdjMatrix //邻接矩阵

{

elementType VerList[MaxVerNum + 1]; //顶点数组，存放顶点元素的值，Data[0]单元不用

cellType AdjMatrix[MaxVerNum + 1][MaxVerNum + 1]; //邻接矩阵，数组下标为0单元不用，从AdjMatrix[1][1]单元开始

int VerNum; //顶点数

int ArcNum; //弧（边）数

GraphKind gKind; //图的类型:0-无向图；1-无向网；2-有向图；3-有向网

} Graph; //图的类型名

邻接表

typedef char elementType; //定义图中顶点的数据类型

typedef int eInfoType; //边链表中关于边的信息的数据类型，比如，带权图中可以表示边的权值

//对无权图，1-相邻（有边），0-不相邻（无边）

//对有权图，为边的权值，特别是无穷大。

typedef enum { UDG, UDN, DG, DN } GraphKind; //枚举图的类型--无向图，无向网，有向图，有向网

typedef struct eNode //边链表结点结构

{

int adjVer; //邻接顶点地址，此处为顶点在顶点表中序号，从1开始

eInfoType eInfo; //边链表中表示边的相关信息，比如表的权值

struct eNode\* next; //指向边链表中的下条边

}EdgeNode; //边

typedef struct vNode //顶点表中元素结构

{

elementType data; //存放图中顶点的数据

EdgeNode\* firstEdge; //指向此顶点关联的第一条边的指针，即边链表的头指针

}VerNode;

typedef struct GraphAdjLinkList //邻接表

{

VerNode VerList[MaxVerNum + 1];//存放顶点的顺序表，数组0单元不用

int VerNum; //顶点数

int ArcNum; //弧（边）数

GraphKind gKind; //图的类型:0-无向图；1-无向网；2-有向图；3-有向网

}Graph; //图的类型名

**6.5算法设计（以邻接表为例）**（部分功能函数省略未打印）

1 【算法思想】深度遍历：

首先将所有顶点都初始化为未访问状态。然后选择一个起始顶点verID，从该顶点开始DFS。此连通分量DFS结束后，再检查是否所有连通分量都已经结束，若没有即visited[vID]==false,从当前顶点开始DFS。直到所有连通分量访问完毕。DFS函数中，首先访问该顶点，然后找到它的第一个邻接点，判断是否访问过，若没有则对他进行DFS。若访问过了找他的另外的邻接点。循环进行上面操作，循环退出条件为所有邻接点都已经访问过。

广度遍历：

首先将所有顶点都初始化为未访问状态。然后选择一个起始顶点verID，从该顶点开始BFS。此连通分量BFS结束后，再检查是否所有连通分量都已经结束，若没有即visited[vID]==false,从当前顶点开始BFS。直到所有连通分量访问完毕。BFS需要利用到队列。BFS函数中，首先访问该顶点，接着将其入队，当队列不空的时候，取队头出队，然后找到它的第一个邻接点，判断是否访问过，若没有则访问它，将他入队。若访问过了找他的另外的邻接点。循环进行上面操作，循环退出条件为所有邻接点都已经访问过。

为了便于操作，队列用数组模拟

【算法描述】

//1.深度优先遍历

void DFS(Graph& G, int verID) {

visit(G, verID);

EdgeNode\* p;

p = G.VerList[verID].firstEdge;

while (p) {

if (!visited[p->adjVer]) {

DFS(G, p->adjVer);

}

p = p->next;

}

}

int DFSTraverse(Graph& G, int verID) {

int vID; //顶点编号

int conNum = 0; //记录连通分量数

for (vID = 0; vID <= G.VerNum; vID++) {

visited[vID] = false;

}

DFS(G, verID);

conNum++;

for (vID = 1; vID <= G.VerNum; vID++) {

if (!visited[vID]) {

DFS(G, vID);

conNum++;

}

}

return conNum;

}

//广度优先遍历

void BFS(Graph& G, int verID) {

int u;

EdgeNode\* p;

int Q[100]; //数组模拟队列

int front = 0;

int rear = 0;

visit(G, verID);

Q[++rear] = verID;

while (front != rear) {

u = Q[++front];

p = G.VerList[u].firstEdge;

while (p) {

if (!visited[p->adjVer]) {

visit(G, p->adjVer);

Q[++rear] = p->adjVer;

}

p = p->next;

}

}

}

int BFSTraverse(Graph& G, int verID) {

int vID;

int conNum = 0;

for (vID = 0; vID <= G.VerNum; vID++) {

visited[vID] = false;

}

BFS(G, verID);

conNum++;

for (vID = 1; vID <= G.VerNum; vID++) {

if (!visited[vID]) {

BFS(G, vID);

conNum++;

}

} return conNum;

}

2 【算法思想】改编深度遍历算法，每次头结点调用对弧长加一，注意对于有向图和有向网，它的所有邻接点的个数就是它的边数。对于无向图和无向网，它的所有邻接点的个数除以二才是它的边数。

【算法描述】

//2.求给定图中的边的数目

void DFS1(Graph& G, int verID, int& E) {

visited[verID] = true;

EdgeNode\* p;

p = G.VerList[verID].firstEdge;

while (p) {

E++;

if (!visited[p->adjVer]) {

DFS1(G, p->adjVer, E);

}

p = p->next;

}

}

int Enum(Graph& G, int verID) {

int vID; //顶点编号

int E = 0;

for (vID = 0; vID <= G.VerNum; vID++) {

visited[vID] = false;

}

for (vID = 1; vID <= G.VerNum; vID++) {

if (!visited[vID]) {

DFS1(G, vID, E);

}

}

if (G.gKind == UDN || G.gKind == UDG)

return E / 2;

else

return E;

}

3 【算法思想】设计算法从V0出发深度优先遍历图G，并构造出相应的生成树或生成森林。改编深度遍历，将访问到的路径信息，存储到树的结构中去。

【算法描述】

//3.深度优先遍历图G，并构造出相应的生成树或生成森林

void DFS\_Traverse(Graph G, csNode\*& T, int verID)

{

T = NULL;

csNode\* locat;//此处定义一个定位指针，用来定位当前树的位置

int vID; //顶点编号

for (int i = 1; i <= G.VerNum; i++)

{

visited[i] = false;

}

csNode\* t = new csNode;//这代表一个小树

t->data = G.VerList[verID].data;

t->firstChild = NULL;

t->nextSibling = NULL;

T = t;//若树为空，建立头节点

locat = t;//定位至小树

Dfs(G, verID, locat);//建立小树

for (int i = 1; i <= G.VerNum; i++)

{

if (!visited[i]) {

csNode\* t = new csNode;//这代表一个小树

t->data = G.VerList[i].data;

t->firstChild = NULL;

t->nextSibling = NULL;

locat->nextSibling = t;//若树不空，则是森林，插入右兄弟

locat = t;//定位至小树

Dfs(G, i, locat);//建立小树

}

}

}

4 【算法思想】设计算法从V0出发广度优先遍历图G，并构造出相应的生成树或生成森林。改编广度遍历算法，将访问到的路径信息，存储到树的结构中去。

【算法描述】

//4.广度优先遍历图G，并构造出相应的生成树或生成森林

void BFSForest(Graph G, csNode\*& T, int verID) {

T = NULL;

csNode\* locat;//此处定义一个定位指针，用来定位当前树的位置

for (int i = 1; i <= G.VerNum; i++)

{

visited[i] = false;

}

csNode\* t = new csNode;//这代表一个小树

t->data = G.VerList[verID].data;

t->firstChild = NULL;

t->nextSibling = NULL;

T = t;//若树为空，建立头节点

locat = t;//定位至小树

Bfs(G, verID, locat);//建立小树

for (int i = 1; i <= G.VerNum; i++)

{

if (!visited[i]) {

csNode\* t = new csNode;//这代表一个小树

t->data = G.VerList[i].data;

t->firstChild = NULL;

t->nextSibling = NULL;

locat->nextSibling = t;//若树不空，则是森林，插入右兄弟

locat = t;//定位至小树

Bfs(G, i, locat);//建立小树

}

}

}

5 【算法思想】 对连通网N=(V,E)，仍设S为已选顶点集，U 为未选顶点集，WE为候选边集，TE为已选 边集，根据上面的讨论，我们得到Prim算 法流程如下： 1、初始化：S为空，U=V，WE为空，TE为空； 2、将起点从U移到S； 3、更新候选边集合WE（一端已选，一端未选）； 4、 如果集合U非空，重复执行下列操作： （a）从候选边集WE中选择一条权值最小的 边，比如(v,u)，加入到已选边集TE中。这 里v为“已选一端”，u为“未选一端”。 如果最小权值的边有几条，任选其中一条； （b）把（a）中所选边的另一端顶点，u，加 入到已选顶点集合S中； （c）更新候选边集合WE； n 循环结束后，整个算法结束，此时有： S=V，U为空，WE为空，TE中为最小生成 树的n-1条边。

【算法描述】

//5．实现Prim算法 基于邻接链表的实现

typedef struct minEdgeType//U指的是已选择的顶点集合 V指的是未选择的顶点集合

{

int v; //V-U中当前选中的顶点编号，从1开始。即刚从V-U中选出放到U中的顶点

eInfoType eWeight; //U中某个顶点到V-U中当前顶点v的最小距离

} MinEdgeType;

void Prim(Graph& G, int vID)

{

MinEdgeType minEdges[MaxVerNum + 1];

int i;

int curID; //当前选择顶点编号

eInfoType wAll = 0; //权值总和

InitMinEdges(G, minEdges, vID); //初始化候选边数组

inTree[vID] = true; //标记vID已在生成树上，即集合U中

for (i = 1; i < G.VerNum; i++) //形成生成树

{

curID = GetMinEdge(G, minEdges); //选择V-U中最小边关联的顶点

inTree[curID] = true; //标记curID已选进U中

ChangeMinEdgesWeight(G, minEdges, curID); //修改权值

}

cout << endl; //输出结果

cout << "Prim生成树起始顶点：" << G.VerList[vID].data << "，编号：" << vID << endl;

cout << "选择的边和权值：" << endl;

for (i = 1; i <= G.VerNum; i++)

{

if (i != vID)

{

cout << "(" << G.VerList[i].data << "," << G.VerList[minEdges[i].v].data << ") 权值：" << minEdges[i].eWeight << endl;

wAll += minEdges[i].eWeight;

}

}

cout << "生成树总权值：" << wAll << endl;

cout << endl;

}

6 【算法思想】实现Kruskal算法，求解下列给定图G的最小生成树。先将图的所有边都保存下来，并且都初始化为可用。还要记录每个连通分量的编号。从所有边中选出最小的可用的边，选完之后将最小边标记为不可用。将选出的边放到生成树边的数组中去。将连通编号更新为最小的编号。直到选出n-1条边，循环结束，最小生成树的边全部找到。

【算法描述】

//6.Kruskal 算法--基于邻接链表

typedef struct edgetype

{

int vBegin; //边的起始顶点编号，从1开始

int vEnd; //边的另一顶点编号，从1开始

eInfoType eWeight; //边的权值

}EdgeType;

//Kruskal算法

void Kruskal(Graph& G)

{

int conVerID[MaxVerNum]; //存放连通分量（子树）编号

EdgeType edges[MaxVerNum \* MaxVerNum]; //存放所有的边信息

EdgeType treeEdges[MaxVerNum - 1]; //存放生成树的所有边信息，共n-1条边

int edgeUsed[MaxVerNum \* MaxVerNum]; //与edges[]数组对应，标记一条边是否已经使用过。1--已用过，0--未用过

//也可以用排序算法先对图的所有边进行排序来完成这个工作。

EdgeType minEdge; //保存最小边

int i, j;

int n; //返回的最小边的序号

int conID; //获取连通分量编号

int M; //循环次数

if (G.gKind == UDG || G.gKind == UDN)

M = G.ArcNum \* 2; //因为无向图、无向网邻接矩阵对称，有效数据是边数的2倍，所以乘2

else

M = G.ArcNum; //有向图或有向网，M=边数

//获取图所有边的信息，存入数组edges[]

GetEdges(G, edges);

for (i = 1; i <= M; i++)

edgeUsed[i] = 0; //标记edges[]中所有边都可用。

//初始化连通分量编号。开始每个顶点作为一个连通分量，每个一个编号，从1开始，与顶点编号相同

for (i = 1; i <= M; i++)

{

conVerID[i] = i;

}

for (i = 1; i < G.VerNum; i++) //取出n-1条边，构成生成树

{

minEdge = GetMinEdge(G, edges, edgeUsed, n); //取得本轮循环的最小边

while (conVerID[minEdge.vBegin] == conVerID[minEdge.vEnd]) //当前最小边2个顶点已经属于同一个连通分量，不可用，继续取下一条最小边、

{

edgeUsed[n] = 1; //标记边edges[n]

minEdge = GetMinEdge(G, edges, edgeUsed, n); //继续取下一条最小边

}

//取得有效最小边，加入最小生成树中

treeEdges[i] = minEdge;

conID = conVerID[minEdge.vBegin]; //取得此最小边开始顶点的连通编号

for (j = 1; j <= G.VerNum; j++)

{

if (conVerID[j] == conID)

conVerID[j] = conVerID[minEdge.vEnd];

}

edgeUsed[i] = 1; //当前最小边标记为已使用边

}

//输出结果

eInfoType wAll = 0; //总权值

cout << endl; //输出结果

cout << "Kruskal最小生成树:" << endl;

cout << "选择的边和权值：" << endl;

for (i = 1; i < G.VerNum; i++) //n-1条边

{

cout << "(" << G.VerList[treeEdges[i].vBegin].data << "," << G.VerList[treeEdges[i].vEnd].data << ") 权值：" << treeEdges[i].eWeight << endl;

wAll += treeEdges[i].eWeight;

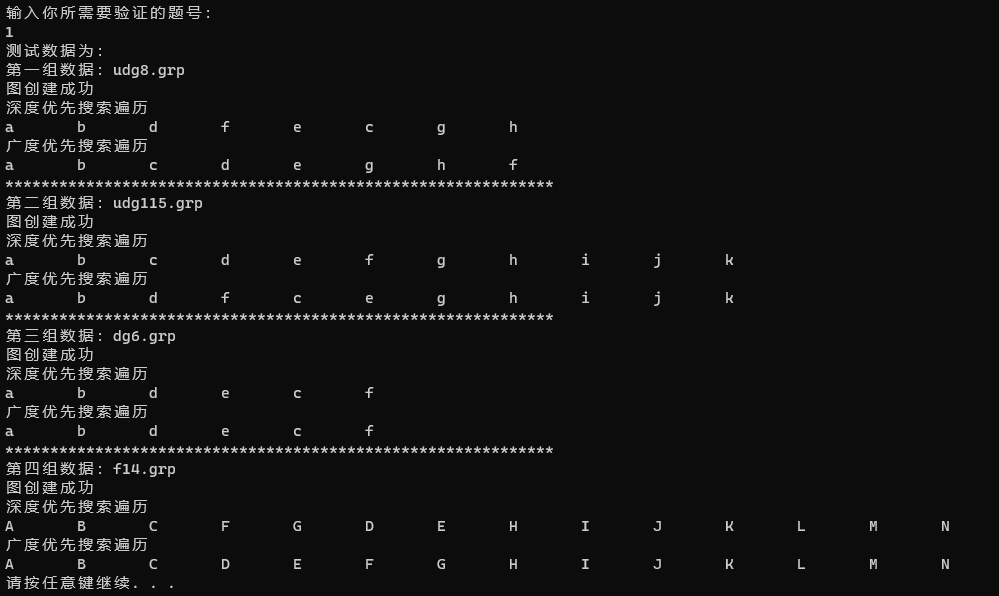
}

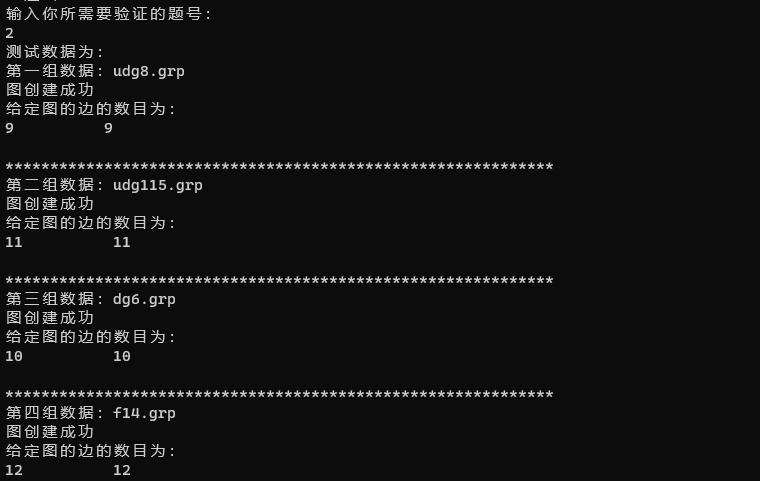
cout << "生成树总权值：" << wAll << endl;

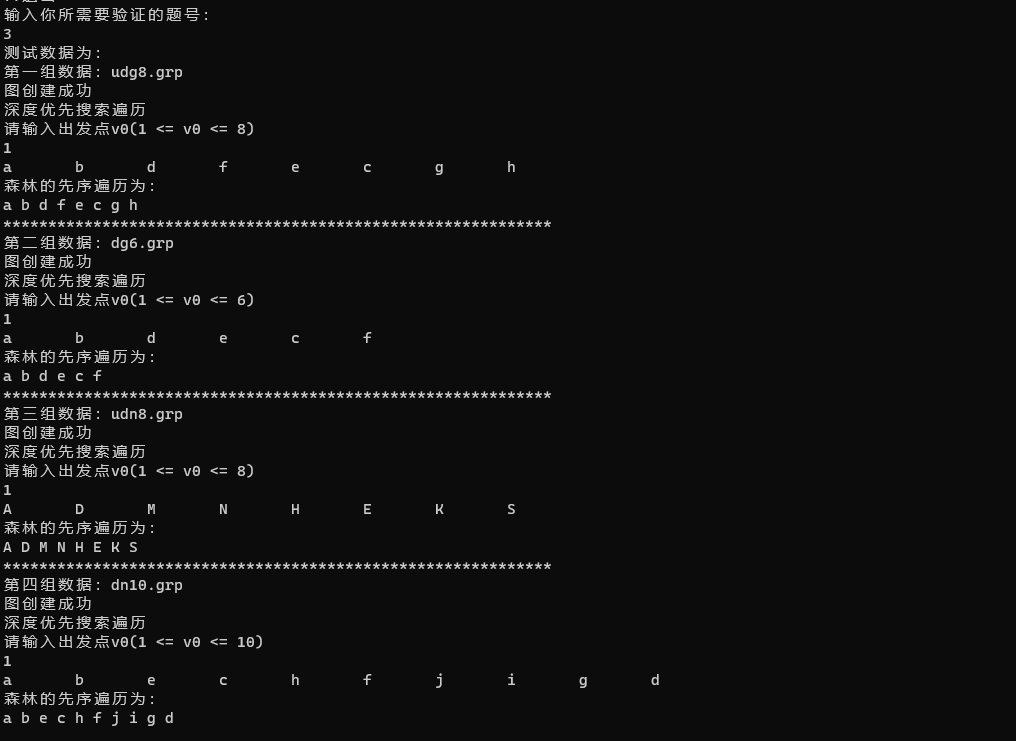
cout << endl;

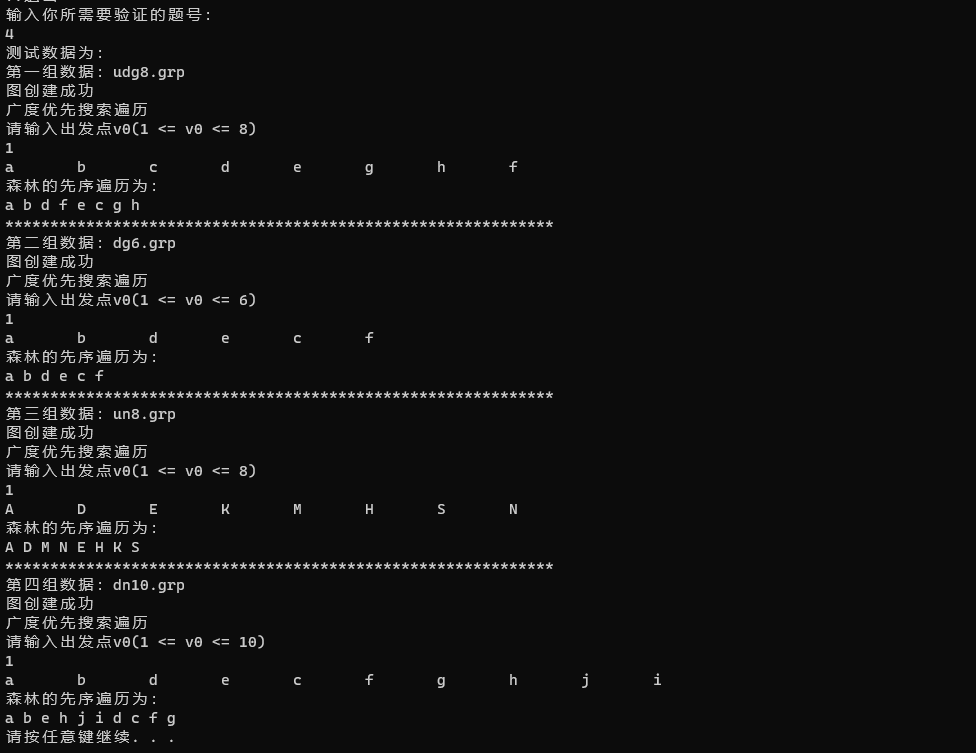
}

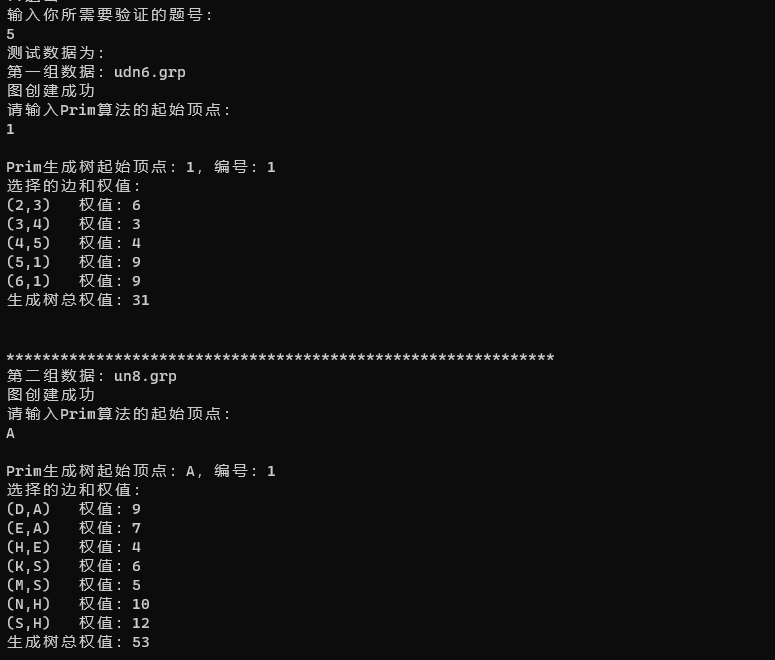
**6.6运行和测试**

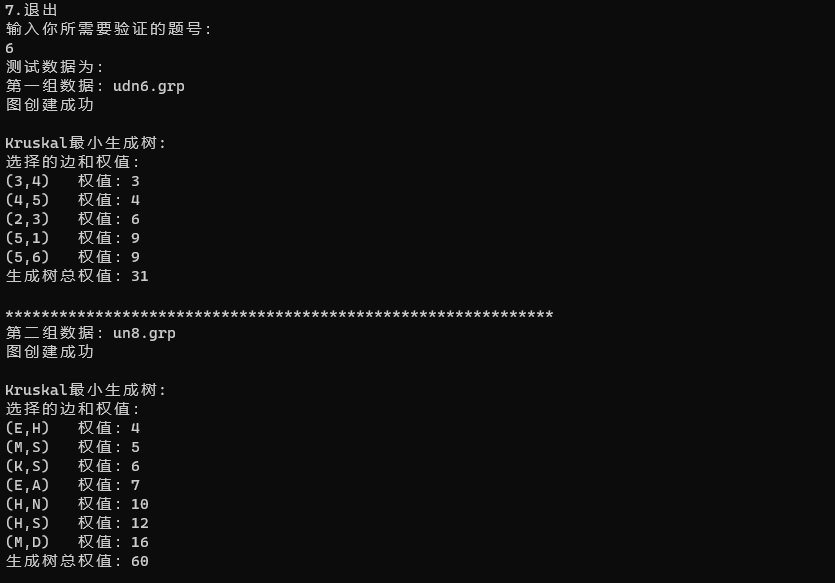












**6.7总结、心得和建议**

在这次图的实验中，我初步掌握了图的基本概念和基本算法，并且收获了以下几点体会和思考。

在图的广度优先遍历和深度优先遍历的实验过程中，我发现不同的遍历算法适用于不同的应用场景。广度优先遍历可以用于查找最短路径，而深度优先遍历则可以用于查找所有可能路径。在实际的应用中，要选择合适的遍历算法，以求得更好的效果。在广度优先遍历时我没有使用STL的queue,而是通过模拟队列实现，拓宽了自己的思维。在调试时由于有一个地方的参数错误，花费了大量时间查找修改，让我深刻的意识到写代码必须认真严谨，全面思考。

在普利姆算法和克鲁斯卡尔算法的实验过程中，我发现两种算法对于同一张图的处理结果是不同的，这证明了算法的局限性和不同算法的优缺点。普利姆算法在求解稠密图的最小生成树问题时更加高效，而克鲁斯卡尔算法在求解稀疏图的最小生成树问题时更加适用，两种算法的防止重复问题也有不同的对策。在实际应用中，也应根据实际问题情况选择最适合的算法。

在本次实验中，我也意识到数据结构的学习仅仅停留于书本知识远远不够，必须将所学知识应用于实际场景进行巩固和实践。通过本次图的实验，我不仅掌握了图数据结构的相关知识和算法，也进一步加深了对数据结构整体框架的认识。

本次图的实验让我获益匪浅。在接下来的学习中，我还将继续深入了解数据结构的有关知识，增强算法能力，将所学知识更好地应用于实际问题中，为未来的学习和工作打下坚实的基础。

**[6.8附录]**

（源代码清单。纸质报告不做要求。电子版报告，可直接附源文件，删除编译生成的所有文件）

****（邻接矩阵 需要对应的头文件编译才能运行）

****（邻接表 需要对应的头文件编译才能运行）