数据结构实验报告

实验成绩：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 专业班级 |  |
| 指导教师 | 张先宜 | 实验时间 | 6月10日 | 实验地点 | 计算机楼 |

**实验名称： 图实验一**

**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**

（所有存储结构的封装描述）

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; //指向边链表中的下一个结点

}EdgeNode; //边链表结点类型

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

{

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

typedef struct GraphAdjMatrix

{

elementType Data[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

} Graph1; //图的类型名

//深度优先遍历图G，并构造出相应的生成树或生成森林

typedef struct csNode

{

elementType data;

struct csNode \*firstChild, \*nextSibling;

}csNode,\*csTree;

//Prim算法基于邻接链表的实现

int inTree[MaxVerNum+1]={0}; //标记顶点已经在Prim生成树中，或已经访问过。1或true--已访问，0或false--未访问

//inTree[0]单元未用

//或者为标记已经在集合U中的顶点

//保存候选边的信息

typedef struct minEdgeType

{

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

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

} MinEdgeType;

//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;

**6.5算法设计**

1．打印出图（网）的两种遍历序。

【算法思想】

（1）假定图 G 是连通的，选定从顶点 v0 出发深度优先搜索遍历算法 DFS(v0)描述如下：

①访问 v0；

②依次从 v0 的各个未被访问的邻接点出发执行深度遍历（DFS）。

(2)如果图 G 是非连通的，在 DFS(G,v)执行结束后，在未被访问的顶点中选一个再次执行 DFS，反复执行这个过程，直到所有顶点都被访问。

//深度遍历

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.求给定图中的边（或弧）的数目。

【算法思想】

① 与 DFS 类似，同样要设访问标志数组 visited。

② 为了能依次访问上一层次的访问序列中的各顶点的邻接点，设置一个结构來保存上一层次的顶点，

（2）一般图的（通用）广度优先搜索遍历算法

与 DFS 算法相似，BFS(G, v)只能从指定顶点 v 出发遍历连通图，或一个连通分量。使用深度遍历相同的处理方法，得到基于 BFS 的遍历整个图的算法

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.深度优先遍历树和深度优先遍历森林算法。

【算法思想】

DFSTree()按照Dfs()深度遍历相应递归生成孩子-兄弟表示的二叉树。而DFS\_Traverse ()则分别处理每棵树，进而转化成森林（二叉链表表示）

void Dfs(Graph G,int i,csNode \*&T)

{

visit(G, i);

bool first=true;//表示是否为当前节点第一个孩子

csNode \*locat;//同样是定位作用

EdgeNode \*p;

p = G.VerList[i].firstEdge;

while(p)//从此节点出发，访问邻接节点。

{

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

csNode \*t=new csNode;//建立一颗小树

t->data=G.VerList[p->adjVer].data;

t->firstChild = NULL;

t->nextSibling = NULL;

if(first){//是当前节点第一个孩子

T->firstChild = t;

first=false;//表示不是传进来的第一个孩子,则是孩子们的兄弟

}else{

locat->nextSibling=t; //建立右孩子

}

locat=t;

Dfs(G,p->adjVer,t);//继续对小树找兄弟

}

p = p->next;

}

}

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.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。

【算法思想】

修改广度优先遍历树和广度优先遍历森林算法。BFSTree()按照Bfs()广度遍历相应分层生成孩子-兄弟表示的二叉树。而BFSForest()则分别处理每棵树，进而转化成森林（二叉链表表示）。

void Bfs(Graph G,int i,csNode\*T){

csNode \*q=NULL;

EdgeNode \*p;

csNode \* Q[100]; // 数组模拟队列

int front = 0;

int rear = 0;

visit(G, i);

//根结点入队

Q[++rear] =T;

//当队列为空时，证明遍历完成

while (front != rear) {

bool first=true;

//队列首个结点出队

q = Q[++front];

//遍历以出队结点为起始点的所有邻接点

int u = LocateVertex(G, q->data);

p = G.VerList[u].firstEdge;

while(p){

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;

//当前结点入队

Q[++rear] = t;

//更改标志位

visit(G, p->adjVer);

//如果是出队顶点的第一个邻接点，设置p结点为其左孩子

if(first){//是当前节点第一个孩子

T->firstChild = t;

first=false;//表示不是传进来的第一个孩子,则是孩子们的兄弟

}else{

q->nextSibling=t; //建立右孩子

}

q=t;

}

p = p->next;

}

}

}

//4.广度优先搜索生成森林并转化为二叉树

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.实现Prim算法，求解下列给定图G的最小生成树。

【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， 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；

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；

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。

void Prim(Graph &G, int vID)

{

MinEdgeType minEdges[MaxVerNum]; //minEdges[i]的下标加1，即i+1为选定边的起始点

//minEdges[i].v为选定边的终点

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++) //选择n-1条边，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[minEdges[i].v].data<<","<<G.VerList[i].data<<") 权值："<<minEdges[i].eWeight<<endl;

wAll+=minEdges[i].eWeight;

}

}

cout<<"生成树总权值："<<wAll<<endl;

cout<<endl;

}

6.实现Kruskal算法，求解下列给定图G的最小生成树。

【算法思想】

对连通网 N=(V, E)，假定初始时构造一个图 T，T 的顶点集为 V，边集为空，即 T=(V, Ф )，

可见初始时 T 为 n 个连通分量的非连通图，或 n 棵子树的森林，每个连通分量（子树）只有一个顶点。算法思想简要描述如下：

①初始化生成图 T，用此存放生成树。

②while( T 的边数 < n-1)

③从 N=(V, E)的边集中选取当前最短边（u,v）； 标记 E 中边(u,v)已选，不可再用;

如果边(u,v)与 T 中已有边不构成回路，将边(u,v)加入到树 T 中;

//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=0; 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; //顶点编号与数组下标差1

}

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]（从0开始）不可用

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;

}

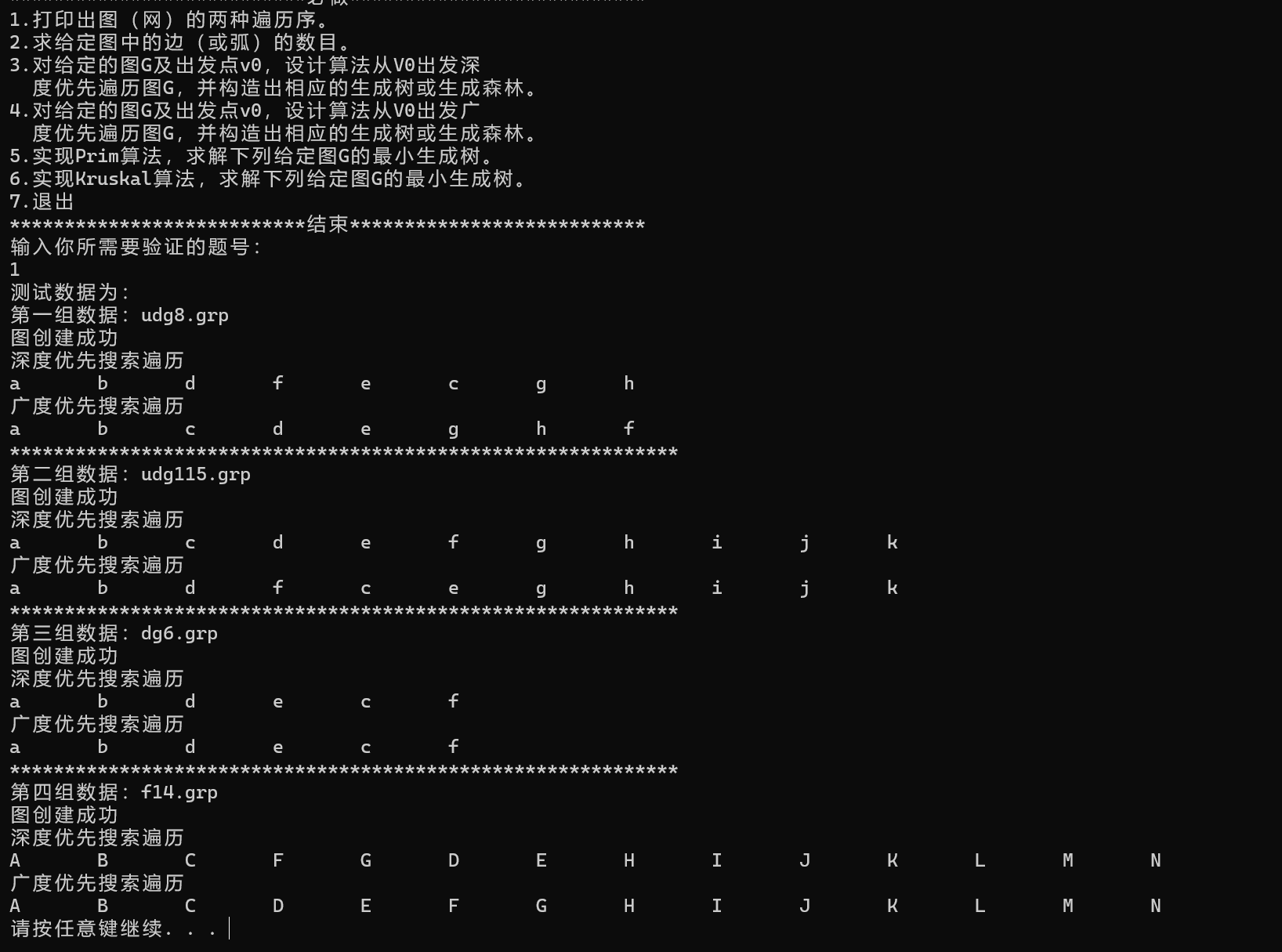
cout<<"生成树总权值："<<wAll<<endl;

cout<<endl;

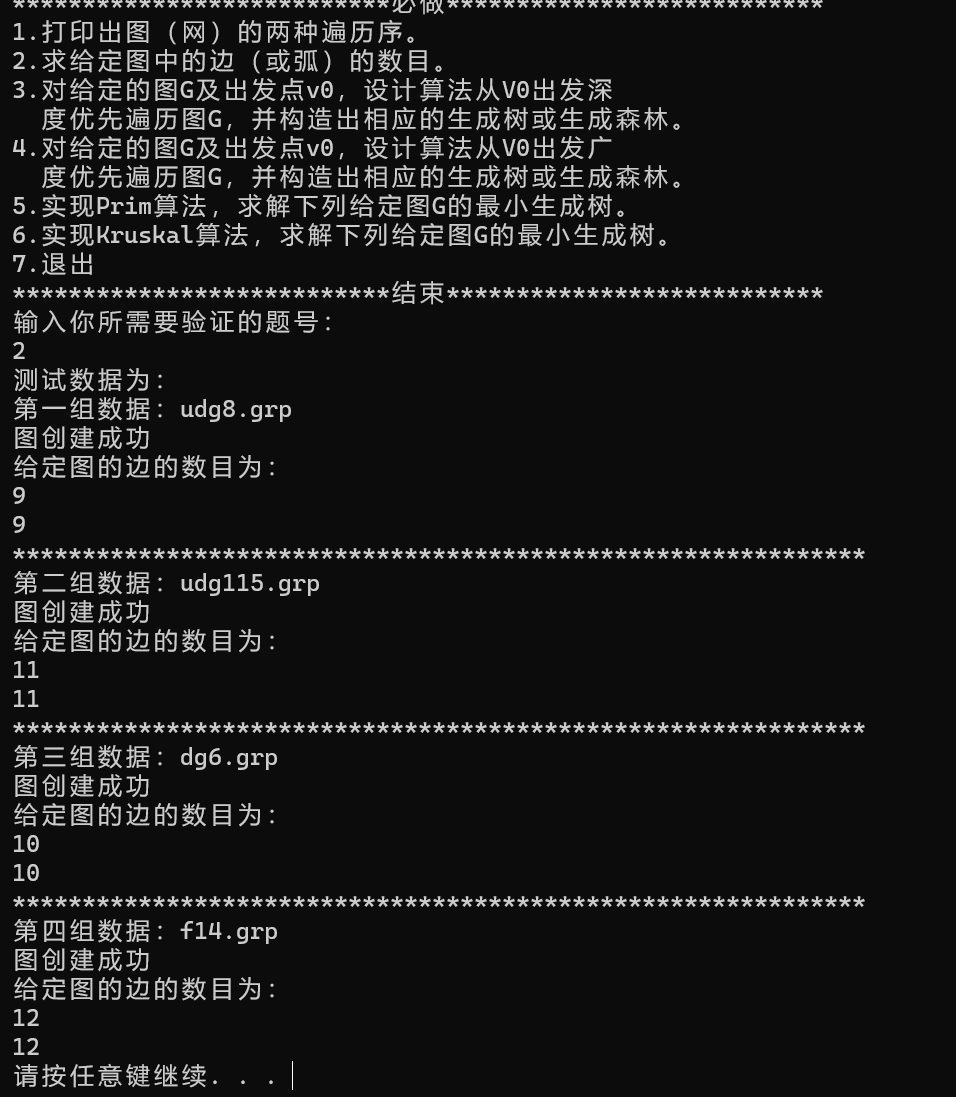
}

**6.6运行和测试**

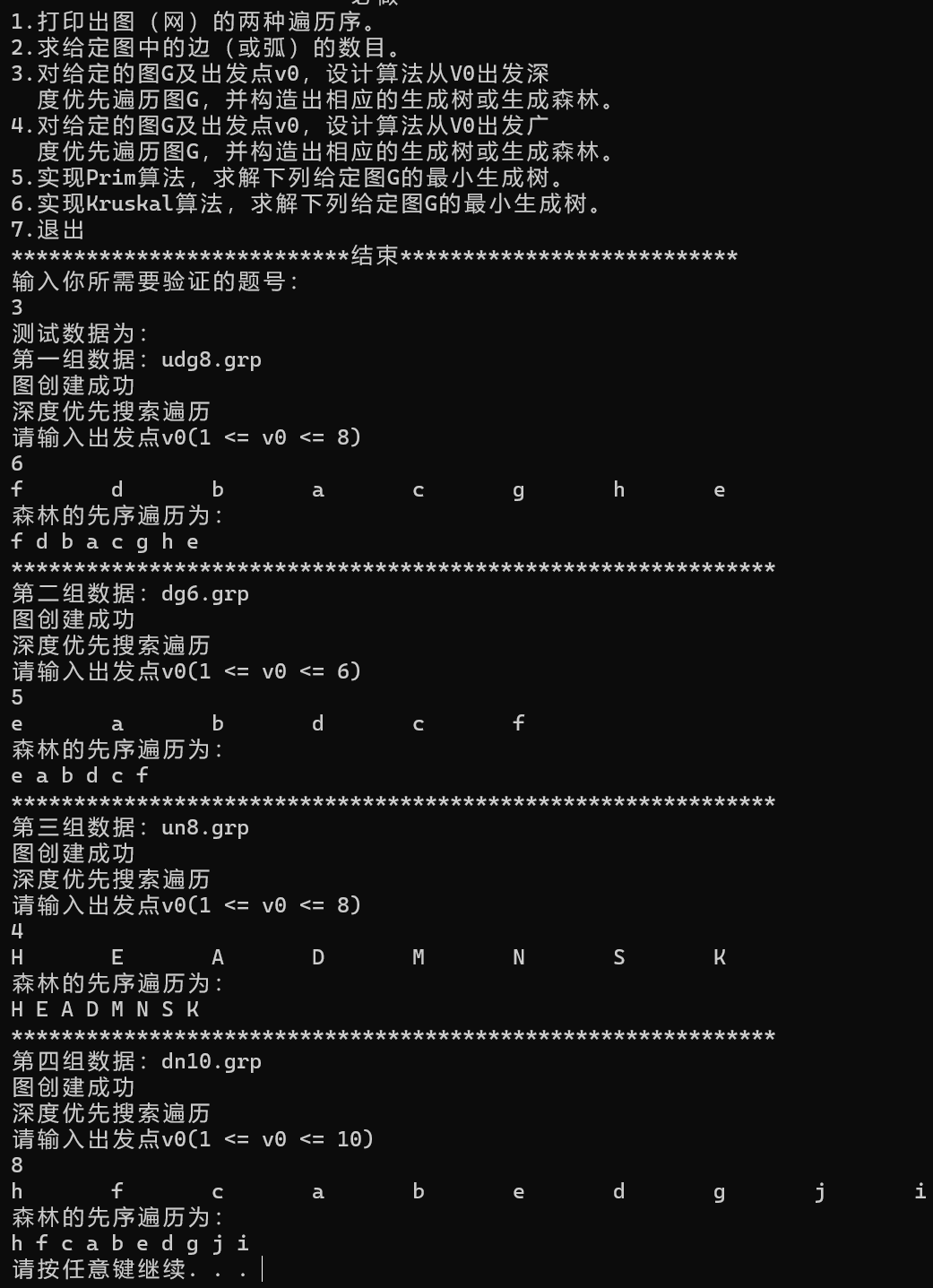
1)．打印出图（网）的两种遍历序。



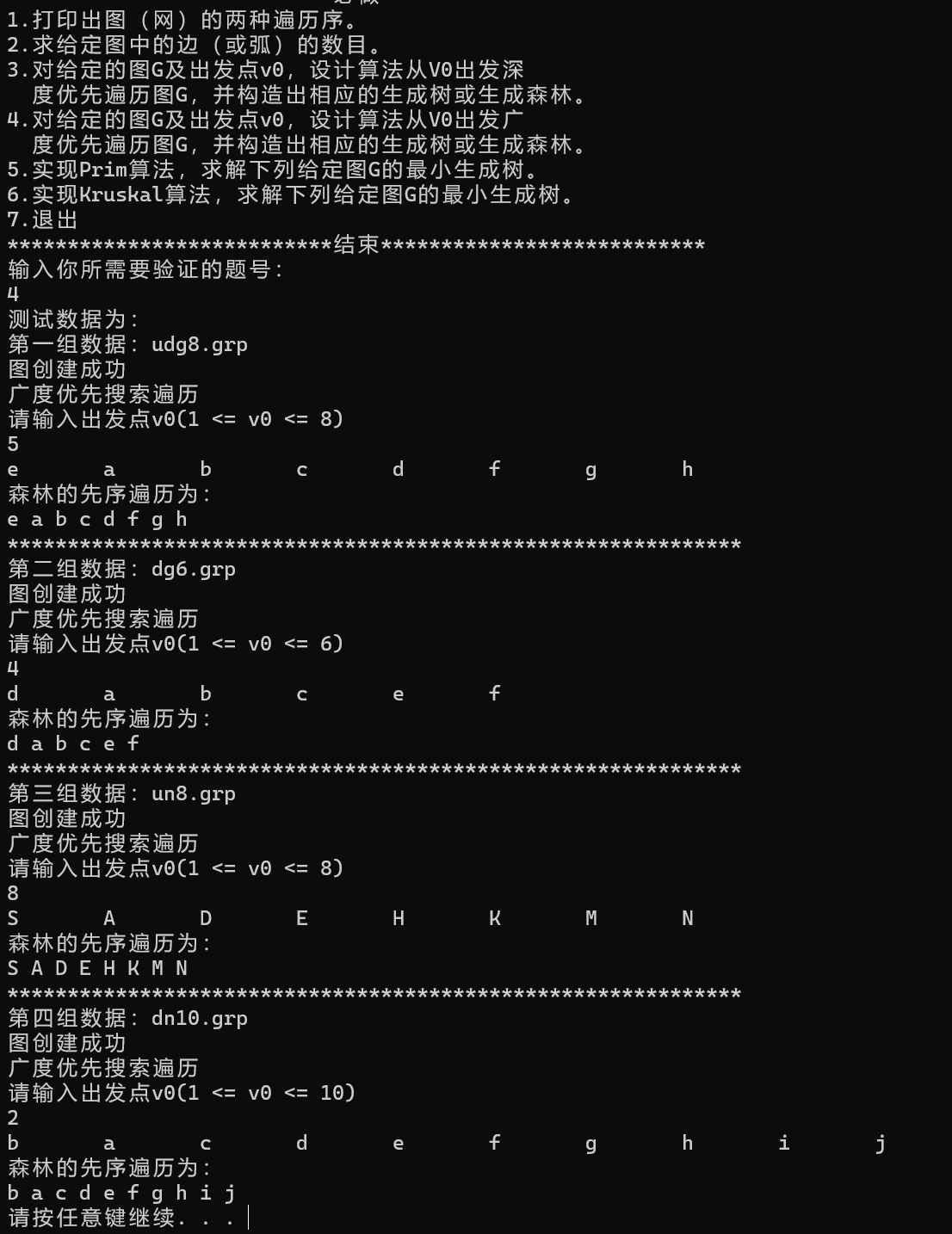
2) 求给定图中的边（或弧）的数目。



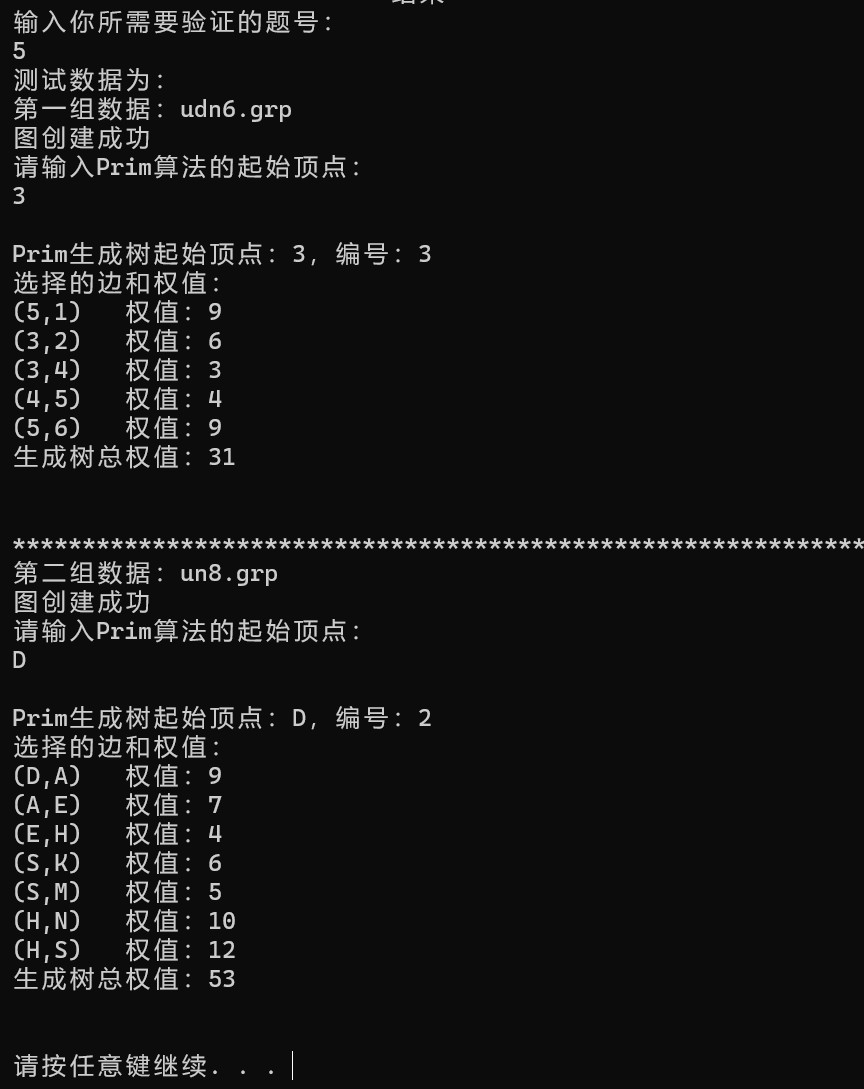
3) 对给定的图G及出发点v0，设计算法从V0出发深度优先遍历图G，并构造出相应的生成树或生成森林。



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。



5) 实现Prim算法，求解下列给定图G的最小生成树。



6) 实现Kruskal算法，求解下列给定图G的最小生成树。



**6.7总结、心得和建议**

图结构是目前数据结构实验中最复杂的实验，所以，图的实验较其它实验来说也比较复杂，但此次图实验基本为书上已有的算法。此次实验主要是对图的基本性质及结构加深理解，对书上的相关图的算法理解并运用。

经过此次实验，我学到了了两种的图常用的存储结构，此次实验主要运用邻接矩阵存储结构。更加体会了图的基本遍历的重要性以及图的广泛运用。关于不同的图问题，有不同不求解算法，书中只是给出了最常用的几种算法，既是基础的算法，在这次试验中，通过对书本的理解以及对老师给出的程序的学习，加深了我对图的理解。