**江西师范大学计算机信息工程学院学生实验报告（8）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **专业：** | **数据科学与大数据技术2班** | **姓名：** | **赖丽婷** | **学号：** |  | **日期：** | **2021.11.24** |

|  |  |  |  |
| --- | --- | --- | --- |
| 课程名称 | 数据结构 | 实验室名称 | 计算机综合实验室 |
| 实验名称 | 图 | | |
| 指导教师 |  | 成绩 |  |

**1.实验目的**（结出本次实验所涉及并要求掌握的知识点）

1. 熟练掌握图的邻接矩阵与邻接表存储结构及运用
2. 能设计出基于两种遍历算法的想关问题的求解，如深度遍历生成树的求解，广度遍历生成树的求解
3. 掌握最小生成树算法的基本思想及其计算方法
4. 理解并掌握最短路径算法的基本思想及其算法方法。
5. 理解并掌握拓扑排序算法的基本思想及其算法方法。

**2.实验内容**（结出实验内容具体描述）

1. 输出无向图各顶点的度
2. 对邻接表图进行广度优先遍历
3. 对图进行深度优先遍历
4. 实现prim求解最小生成树算法
5. 实现Dijkstra求单源最短路径算法
6. 实现图的拓扑排序

**3.算法描述及实验步骤**（用适当的形式表达算法设计思想与算法实现步骤）

1. 第一步建立一个用于记录各顶点的数组，第二步遍历邻接表，通过各顶点的连接关系确定各顶点的度，
2. 第一步，建立一个队列，第二步，遍历邻接表，选定一个顶点，将该顶点进队列，然后访问该顶点，然后将该点出队列，并记下该顶点的位置，用此位置来将该顶点的所有未访问的连接点进队列，第三步，第二步结束后，队首出队列，然后访问该顶点，然后将该点出队列，并记下该顶点的位置，用此位置来将该顶点的所有连接点进队列。
3. 第一步，选定一个顶点，访问该顶点的值，第二步，访问该顶点的第一个连接点，第三步，进行递归，找到当前顶点的第一个未被访问的结点，打印出来，直到将整张图遍历完成
4. 最小生成树的prim算法我们知道每次在原图中找到最短的边，对于顶点u属于U,v属于V-U，我们找到u-v最短的边，然后把v加入U直到把所有顶点都加入。

**初始化：**

我们先设一个顶点v0到其他顶点的权值为初始值，当加入新顶点v后，如果v-vi的权值小于v0到vj的权值，则用v-vi代替v0-vi，后面节点加入后同理。

1. 对于Dijkstra算法，将所有顶点分为两组，一组是已经确定最短路径的顶点，另一组为未确定最短路径的点，然后去寻找到未确定最短路径的点的距离的最小值，然后加入到确定组中。

我们用数组d来保证路径长度，用p来记录路径。

**算法实现：**

将原点放入S中，并将其到其他顶点的距离作为长度，如果没有路则为定义的无穷大。

从V-S中选出到S最小的顶点设为v加入S,然后对新加入的顶点把距离修改，如果v0-v到V-S(设为k)的距离小于v0-k，则距离替换为更小的一方。

如果V-S无顶点，则结束

1. 拓扑排序的思路是遍历图，当其入度为0时输出，并将其邻接点的入度减1，直到顶点全部输出。

**具体算法：**

修改邻接表的结构，加入一块内存用于存放顶点的入度，在此过程中需要用一个flag来判断是否被访问过。

将所有的入读为0的顶点进队列，并当队列不空时，取出队首输出，并将其邻接点入度-1，如果邻接点-1后入度便为0则进队列。

当对列为空时则结束。

**4.调试过程及运行结果**（详细记录在调试过程中出现的问题及解决方法。记录实验执行的结果）

**5. 总结**（对实验结果进行分析，问题回答，实验心得体会及改进意见）

图的生成与遍历操作虽然比之前学的顺序表与链式表麻烦，但是核心还是不变的，只是图的结构更加麻烦一些，图的最小生成树问题以及最短路径问题还是挺难的，感觉学的不精还需要多多练习

**6.附录**（程序源代码等）

#include "ljb.h"

/\* 输出以邻接表为存储结构的无向图g的各顶点的度 \*/

void degree(LinkedGraph g)

{

int i, j;

int degreenum[M];

EdgeNode \*p;

for(i = 0; i < g.n; i++){

p = g.adjlist[i].FirstEdge;

j = 0;

while(p){

degreenum[i] = ++j;

p = p->next;

}

}

for(i = 0; i < g.n; i++){

printf("%d 's degree is %d\n",i ,degreenum[i]);

}

}

int main()

{

LinkedGraph g;

creat(&g, "../g11.txt", 0); /\*已知g11.txt中存储了图的信息\*/

printf("\n The graph is:\n");

print(g);

degree(g);

return 0;

}

#include "ljb.h"

int visited[M]; /\*全局标志向量\*/

/\*请将本函数补充完整，并进行测试\*/

void bfs(LinkedGraph g, int i)

{ /\*从顶点i出发广度优先变量图g的连通分量\*/

int j;

EdgeNode \*p;

int quence[M], front, rear;

front = rear = 0;

printf("the first node is %c\n", g.adjlist[i].vertex);

visited[i] = 1;

quence[rear++] = i;

while (front < rear)

{

j = quence[front++];

p = g.adjlist[j].FirstEdge;

while (p)

{

if (visited[p->adjvex] == 0)

{

printf("node is %c\n", g.adjlist[p->adjvex].vertex);

quence[rear++] = p->adjvex;

visited[p->adjvex] = 1;

}

p = p->next;

}

}

}

/\*函数功能：广度优先遍历图g

函数参数：邻接表g

\*/

int BfsTraverse(LinkedGraph g)

{

int i, count = 0;

for (i = 0; i < g.n; i++)

visited[i] = 0; /\*初始化标志数组\*/

for (i = 0; i < g.n; i++)

if (!visited[i]) /\*vi未访问过\*/

{

printf("\n");

count++; /\*连通分量个数加1\*/

bfs(g, i);

}

return count;

}

int main()

{

setbuf(stdout,NULL);

LinkedGraph g;

int count;

creat(&g, "../g11.txt", 0); /\*创建图的邻接表\*/

printf("\n The graph is:\n");

print(g);

printf("广度优先遍历序列为：\n");

count = BfsTraverse(g); /\*从顶点0出发广度优先遍历图g\*/

printf("\n该图共有%d个连通分量。\n", count);

return 0;

}

#include "ljb.h"

int visited[M];

/\*请将本函数补充完整，并进行测试\*/

void dfs(LinkedGraph g, int i)

{ /\*从顶点i开始深度优先遍历图的连通分量\*/

EdgeNode \*p;

printf("visit vertex: %c \n", g.adjlist[i].vertex);/\*访问顶点i\*/

visited[i] = 1;

p = g.adjlist[i].FirstEdge;

while (p) /\*从p的邻接点出发进行深度优先搜索\*/

{

if(visited[p->adjvex] == 0)

{

dfs(g, p->adjvex);

}

p = p->next;

}

}

/\*函数功能：深度优先遍历图

函数参数：图的邻接表g

\*/

void DfsTraverse(LinkedGraph g)

{

int i;

for (i = 0; i < g.n; i++)

visited[i] = 0; /\*初始化标志数组\*/

for (i = 0; i < g.n; i++)

if (!visited[i]) /\*vi未访问过\*/

dfs(g, i);

}

int main()

{

LinkedGraph g;

creat(&g, "../g11.txt", 0); /\*创建图的邻接表\*/

printf("\n The graph is:\n");

print(g);

printf("深度优先遍历序列为：\n");

DfsTraverse(g); /\*从顶点0开始深度优先遍历图无向图g\*/

}

void prim(Mgraph g, edge tree[M-1])

{ edge x;

int d,min,j,k,s,v;

/\* 建立初始入选点，并初始化生成树边集tree\*/

for (v=1;v<=g.n-1;v++)

{

tree[v-1].beg=0;

tree[v-1].en=v;

tree[v-1].length=g.edges[0][v];

}

/\*依次求当前(第k条）最小两栖边，并加入TE\*/

for (k=0;k<=g.n-3;k++)

{

min=tree[k].length;

s=k;

for (j=k+1;j<=g.n-2;j++)

if (tree[j].length<min)

{

min=tree[j].length;

s=j;

}

v=tree[s].en;

x=tree[s];

tree[s]=tree[k];

/\*由于新顶点v的加入，修改两栖边的基本信息\*/

for (j=k+1;j<=g.n-2;j++)

{

d=g.edges[v][tree[j].en];

if (d<tree[j].length)

{

tree[j].length=d;

tree[j].beg=v;

}

}

}

/\*输出最小生成树\*/

printf("\n最小生成树是:\n");/\*输出最小生成树\*/

for (j=0;j<=g.n-2;j++)

printf("\n%c---%c %d\n",g.vexs[tree[j].beg],g.vexs[tree[j].en],tree[j].length);

printf("\n最小生成树的根是： %c\n", g.vexs[0]);

}

void dijkstra(Mgraph g,int v0,path p,dist d)

{ boolean final[M]; /\*表示当前元素是否已求出最短路径\*/

int i,k,j,v,min,x;

/\* 第1步 初始化集合S与距离向量d \*/

for (v=0;v<g.n;v++)

{

final[v]=*FALSE*;

d[v]=g.edges[v0][v];

if (d[v]<FINITY &&d[v]!=0)

p[v]=v0; else p[v]=-1;

}

final[v0]=*TRUE*;

d[v0]=0;

/\* 第2步 依次找出n-1个结点加入S中 \*/

for (i=1;i<g.n;i++)

{

min=FINITY;

for (k=0;k<g.n;k++)

if (!final[k] && d[k]<min)

{

v=k;

min=d[k];

}

if (min==FINITY) return ;

final[v]=*TRUE*;

/\*第3步 修改S与V-S中各结点的距离\*/

for (k=0;k<g.n;++k)

if (!final[k] && (min+g.edges[v][k]<d[k]))

{

d[k]=min+g.edges[v][k];

p[k]=v;

}

}

}

int TopSort(AovGraph g)

{int k=0,i,j,v, flag[M];

int queue[M]; /\*队列\*/

int h=0,t=0;

edgenode\* p;

for (i=0;i<g.n;i++) flag[i]=0; /\*访问标记初始化\*/

/\*先将所有入度为0的结点进队\*/

/\*将程序补充完整\*/

for (i=0;i<g.n;i++)

if (g.adjlist[i].id==0)

{

queue[t++]=i;

flag[i]=1;

}

while (h<t)

{

v=queue[h++];

printf("%c",g.adjlist[v].vertex);

k++;

p=g.adjlist[v].FirstEdge;

while (p)

{

j=p->adjvex;

g.adjlist[j].id--;

if (g.adjlist[j].id==0 && flag[j]==0)

{ queue[t++]=j;

flag[j]=1;

}

p=p->next;

}

}

return k; //返回输出的顶点个数