重庆大学本科学生实验项目任务书

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验题目 | 图的操作 | | | |
| 实验时间 | 2021. 06. 04 | 实验地点 | | DS1401 |
| 实验性质 | □验证性 √设计性 □综合性 | | | |
| 实验目的  1. 掌握图的基本概念、表示方法、遍历方法。  2. 掌握图的最短路径算法。 | | | | |
| 实验内容：  1． 输入图的顶点数n（不超过10个）、边数m，顶点分别用0到n-1表示；  2． 采用“起始顶点，终止顶点，权值”输入图的m条边，创建图；  3． 输出从顶点0开始的BFS遍历、DFS遍历的顶点遍历顺序；  4． 输出从顶点0到其余顶点最短路径及最短路径的长度，如果没有路经，输出0。 | | | | |
| 参考资料：   * + 《数据结构（ C 语言版）》，严蔚敏，吴伟民编著，清华大学出版社，2007年第1版 | | | | |
| 任务下达日期 2021 年 5 月 25 日 | | | 完成日期 2021 年 6 月 4 日 | |

1. 源程序代码：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* 图的宏定义 \*/

#define INFINITY INT\_MAX // 最大值∞

#define MAX\_VERTEX\_NUM 20 // 最大顶点个数

/\* 状态码 \*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

#define OVERFLOW -2

typedef int Status;

/\* -------辅助队列的存储表示------- \*/

typedef int QElemType;

/\* 队列结点存储表示 \*/

typedef struct QNode{

QElemType data;

struct QNode \*next;

}QNode, \*QueuePtr;

/\* 队列的存储表示 \*/

typedef struct {

QueuePtr front; // 队头指针

QueuePtr rear; // 队尾指针

}LinkQueue;

/\* 初始化队列 \*/

Status InitQueue(LinkQueue \*Q){

// 构造一个空队列

(\*Q).front = (\*Q).rear = (QueuePtr)malloc(sizeof(QNode));

if (!(\*Q).front) exit(OVERFLOW); // 存储分配失败

(\*Q).front->next = NULL;

return OK;

}// InitQueue

/\* 销毁队列 \*/

Status DestroyQueue(LinkQueue \*Q){

// 销毁队列Q

while ((\*Q).front){

(\*Q).rear = (\*Q).front->next;

free((\*Q).front);

(\*Q).front = (\*Q).rear;

}// while

return OK;

}// DestroyQUeue

/\* 判断队列是否为空 \*/

Status QueueEmpty(LinkQueue Q){

// 判断队列是否为空, 为空则返回1, 非空则返回0

if (Q.front == Q.rear) return TRUE;

else return FALSE;

}// QueueEmpty

/\* 队列插入 \*/

Status EnQueue(LinkQueue \*Q, QElemType e){

// 插入元素e为Q的新的队尾元素

QNode \*p; // p为暂存队列结点

p = (QueuePtr)malloc(sizeof(QNode));

if (!p) exit(OVERFLOW); // 存储分配失败

p->data = e; p->next = NULL; // p为末尾最后一个结点

(\*Q).rear->next = p; // 将p接入队尾

(\*Q).rear = p; // 队尾指针指向p

return OK;

}// EnQueue

/\* 队列删除 \*/

Status DeQueue(LinkQueue \*Q, QElemType \*e){

// 若队列不空, 则删除Q的队头元素, 用e返回其值, 并返回OK

// 否则返回ERROR

if ((\*Q).front == (\*Q).rear) return ERROR;

QNode \*p;

p = (\*Q).front->next;

e = p->data;

(\*Q).front->next = p->next;

if ((\*Q).rear == p) (\*Q).rear = (\*Q).front;

free(p);

return OK;

}// DeQueue

/\* -------图的邻接矩阵存储表示------- \*/

typedef int VRType; // 顶点关系类型

typedef int VertexType; // 顶点向量类型

/\* 图的类型的定义 \*/

typedef enum {DG, DN, UDG, UDN} GraphKind; // {有向图, 有向网, 无向图, 无向网}

/\* 顶点的存储结构 \*/

typedef struct ArcCell{

VRType adj; // VRType是顶点关系类型. 对无权图, 用1或0标识相邻否

// 对于带权图, 则表示权值类型.

}ArcCell, AdjMatrix[MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM];

/\* 图的存储结构 \*/

typedef struct {

VertexType vexs[MAX\_VERTEX\_NUM]; // 顶点向量

AdjMatrix arcs; // 邻接矩阵

int vexnum, arcnum; // 图的当前顶点数和弧数

int kind; // 图的种类标志

}MGraph;

/\* 图的顶点定位 \*/

Status LocateVex(MGraph G, VRType u){

// 若图G中存在顶点u, 则返回该顶点在图中的位置; 否则返回其他信息

int j = -1, k;

for (k=0; k<G.vexnum; ++k){

if (G.vexs[k] == u){

j = k;

break;

}// if

}// for

return j;

}// LocateVex

/\* 无向图的构造 \*/

Status CreateUDG(MGraph \*G){

// 采用邻接矩阵的方法, 构造无向网G

// 初始化无向图

printf("请输入无向图的顶点数目和弧的数目:\n");

scanf("%d %d", &(\*G).vexnum, &(\*G).arcnum);

printf("请输入图的顶点向量:\n");

for (int i=0; i<(\*G).vexnum; ++i) scanf("%d", &(\*G).vexs[i]); // 构造顶点向量

for (int i=0; i<(\*G).vexnum; ++i) // 初始化邻接矩阵

for (int j=0; j<(\*G).vexnum; ++j)

(\*G).arcs[i][j].adj = 0; // 设置每条边初始为0

printf("请连续输入弧的起止顶点, 之间用空格隔开\n");

for (int k=0; k<(\*G).arcnum; ++k){

VRType v1, v2; // v1和v2均为图的顶点

// printf("请输入顶点v1、v2和它们之间弧的权值:\n");

scanf("%d %d", &v1, &v2); // 输入一条边依附的顶点的权值

// 确定v1和v2的在G中的位置

int i = LocateVex((\*G), v1);

int j = LocateVex((\*G), v2);

(\*G).arcs[i][j].adj = 1; // 有弧边设为权值1

(\*G).arcs[j][i] = (\*G).arcs[i][j];

// printf("%d %d %d %d", i, j, w, (\*G).arcs[i][j].adj);

}// for

// 打印邻接矩阵

printf("构建成功, 此网的邻接矩阵表示如下:\n");

printf("其中+表示两个顶点之间没有权值.\n");

for (int i=0; i<(\*G).vexnum; ++i){ // 双层for循环打印

printf("%d: ", (\*G).vexs[i]);

for (int j=0; j<(\*G).vexnum; ++j){

if ((\*G).arcs[i][j].adj == INFINITY) printf("%c\t", '+');

else printf("%d\t", (\*G).arcs[i][j].adj);

}// for

printf("\n");

}// for

return OK;

}// CreateUDG

/\* 无向网的构造 \*/

Status CreateUDN(MGraph \*G){

// 采用邻接矩阵的方法, 构造无向网G

// 初始化无向网

printf("请输入无向图的顶点数目和弧的数目:\n");

scanf("%d %d", &(\*G).vexnum, &(\*G).arcnum);

printf("请输入网的顶点向量:\n");

for (int i=0; i<(\*G).vexnum; ++i) scanf("%d", &(\*G).vexs[i]); // 构造顶点向量

for (int i=0; i<(\*G).vexnum; ++i) // 初始化邻接矩阵

for (int j=0; j<(\*G).vexnum; ++j)

(\*G).arcs[i][j].adj = INFINITY; // 设置每条边初始为无穷大

printf("请连续输入起止顶点和他们之间的权值, 权值之间用空格隔开\n");

for (int k=0; k<(\*G).arcnum; ++k){

VRType v1, v2; // v1和v2均为图的顶点

int w; // w为顶点是v1和v2的弧的权值

// printf("请输入顶点v1、v2和它们之间弧的权值:\n");

scanf("%d %d %d", &v1, &v2, &w); // 输入一条边依附的顶点的权值

// 确定v1和v2的在G中的位置

int i = LocateVex((\*G), v1);

int j = LocateVex((\*G), v2);

(\*G).arcs[i][j].adj = w; // 弧<v1, v2>的权值

(\*G).arcs[j][i] = (\*G).arcs[i][j];

// printf("%d %d %d %d", i, j, w, (\*G).arcs[i][j].adj);

}// for

// 打印邻接矩阵

printf("构建成功, 此网的邻接矩阵表示如下:\n");

printf("其中+表示两个顶点之间没有权值.\n");

for (int i=0; i<(\*G).vexnum; ++i){ // 双层for循环打印

printf("%d: ", (\*G).vexs[i]);

for (int j=0; j<(\*G).vexnum; ++j){

if ((\*G).arcs[i][j].adj == INFINITY) printf("%c\t", '+');

else printf("%d\t", (\*G).arcs[i][j].adj);

}// for

printf("\n");

}// for

return OK;

}// CreateDUN

/\* 有向图的构造 \*/

Status CreateDG(MGraph \*G){

// 采用邻接矩阵的方法, 构造无向网G

// 初始化有向图

printf("请输入有向图的顶点数目和弧的数目:\n");

scanf("%d %d", &(\*G).vexnum, &(\*G).arcnum);

printf("请输入图的顶点向量:\n");

for (int i=0; i<(\*G).vexnum; ++i) scanf("%d", &(\*G).vexs[i]); // 构造顶点向量

for (int i=0; i<(\*G).vexnum; ++i) // 初始化邻接矩阵

for (int j=0; j<(\*G).vexnum; ++j)

(\*G).arcs[i][j].adj = 0; // 设置每条边初始为0

printf("请连续输入弧的起止顶点, 之间用空格隔开\n");

for (int k=0; k<(\*G).arcnum; ++k){

VRType v1, v2; // v1和v2均为图的顶点

// printf("请输入顶点v1、v2和它们之间弧的权值:\n");

scanf("%d %d", &v1, &v2); // 输入一条边依附的顶点的权值

// 确定v1和v2的在G中的位置

int i = LocateVex((\*G), v1);

int j = LocateVex((\*G), v2);

(\*G).arcs[i][j].adj = 1; // 有弧边设为权值1

// printf("%d %d %d %d", i, j, w, (\*G).arcs[i][j].adj);

}// for

// 打印邻接矩阵

printf("构建成功, 此网的邻接矩阵表示如下:\n");

printf("其中+表示两个顶点之间没有权值.\n");

for (int i=0; i<(\*G).vexnum; ++i){ // 双层for循环打印

printf("%d: ", (\*G).vexs[i]);

for (int j=0; j<(\*G).vexnum; ++j){

if ((\*G).arcs[i][j].adj == INFINITY) printf("%c\t", '+');

else printf("%d\t", (\*G).arcs[i][j].adj);

}// for

printf("\n");

}// for

return OK;

}// CreateDG

/\* 有向网的构造 \*/

Status CreateDN(MGraph \*G){

// 采用邻接矩阵的方法, 构造无向网G

// 初始化有向网

printf("请输入有向网的顶点数目和弧的数目:\n");

scanf("%d %d", &(\*G).vexnum, &(\*G).arcnum);

printf("请输入网的顶点向量:\n");

for (int i=0; i<(\*G).vexnum; ++i) scanf("%d", &(\*G).vexs[i]); // 构造顶点向量

// printf("%d\n", (\*G).vexnum);

for (int i=0; i<(\*G).vexnum; ++i) // 初始化邻接矩阵

for (int j=0; j<(\*G).vexnum; ++j)

(\*G).arcs[i][j].adj = INFINITY; // 设置每条边初始为无穷大

printf("请连续输入起止顶点和他们之间的权值, 权值之间用空格隔开\n");

for (int k=0; k<(\*G).arcnum; ++k){

VRType v1, v2; // v1和v2均为图的顶点

int w; // w为顶点是v1和v2的弧的权值

// printf("请输入顶点v1、v2和它们之间弧的权值:\n");

scanf("%d %d %d", &v1, &v2, &w); // 输入一条边依附的顶点的权值

// 确定v1和v2的在G中的位置

int i = LocateVex((\*G), v1);

int j = LocateVex((\*G), v2);

(\*G).arcs[i][j].adj = w; // 弧<v1, v2>的权值

// printf("%d %d %d %d", i, j, w, (\*G).arcs[i][j].adj);

}// for

// 打印邻接矩阵

printf("构建成功, 此网的邻接矩阵表示如下:\n");

printf("其中+表示两个顶点之间没有权值.\n");

for (int i=0; i<(\*G).vexnum; ++i){ // 双层for循环打印

printf("%d: ", (\*G).vexs[i]);

for (int j=0; j<(\*G).vexnum; ++j){

if ((\*G).arcs[i][j].adj == INFINITY) printf("%c\t", '+');

else printf("%d\t", (\*G).arcs[i][j].adj);

}// for

printf("\n");

}// for

return OK;

}// CreateDN

/\* 创建图的选择 \*/

Status CreateGraph(MGraph \*G){

// 采用邻接矩阵表示法, 构造图G

scanf("%d", &(\*G).kind);

// CreateDN(&G);

// CreateDN(&(\*G));

// printf("\n%d", (\*G).vexnum);

switch ((\*G).kind){

case 1: CreateDG(&(\*G)); break; // 构造有向图G

case 2: CreateDN(&(\*G)); break; // 构造有向网G

case 3: CreateUDG(&(\*G)); break; // 构造无向图G

case 4: CreateUDN(&(\*G)); break; // 构造无向网G

}// switch

return OK;

}// CreateGraph

/\* 构建访问标志数组 \*/

int visited[MAX\_VERTEX\_NUM];

/\* 图的深度优先搜索初始化 \*/

void DFSTraverse(MGraph G){

// 对图G作深度优先搜索

for (int i=0; i<G.vexnum; ++i) visited[i] = FALSE; // 访问标志数组初始化

for (int i=0; i<G.vexnum; ++i)

if (!visited[i]) DFS(G, i); // 对尚未访问的顶点调用DFS

}// DFSTraverse

/\* 图的深度优先搜索算法 \*/

void DFS(MGraph G, int v){

// 从第v个顶点出发递归地深度优先遍历图G

visited[v] = TRUE; printf("%d ", G.vexs[v]); // 访问第v个顶点, 此算法即打印顶点

for (int i=0; i<G.vexnum; ++i){

if(!visited[i] && G.arcs[v][i].adj!=INFINITY)

DFS(G, i);

}// for

}// DFS

/\* 图的广度优先搜索 \*/

void BFSTraverse(MGraph G){

// 按广度优先非递归遍历图, 使用辅助队列Q和访问标志数组visited

for (int i=0; i<G.vexnum; ++i) visited[i] = FALSE;

LinkQueue Q; int v, u,v0;

InitQueue(&Q); // 置空的辅助队列Q

for (v=0; v<G.vexnum; ++v)

if (!visited[v]){ // v尚未被访问

visited[v] = TRUE;

printf("%d ", G.vexs[v]);

EnQueue(&Q, v); // v入队列

while (!QueueEmpty(Q)){

DeQueue(&Q, &u); // 队头元素出队并置为u

v0 = LocateVex(G, u);

for (int i=0; i<G.vexnum; ++i){

if (!visited[i] && G.arcs[v0][i].adj!=INFINITY){ // 没有在标志数组中且与v0相连

visited[i] = TRUE;

printf("%d ", G.vexs[i]);

EnQueue(&Q, G.vexs[i]); // 将相邻的顶点入队

}// if

}// for

}// while

}// if

}// BFSTraverse

/\* 图的最短路径, 从某一顶点 \*/

/\* 采用Dijkstra算法 \*/

void ShortestPath\_DIJ(MGraph G, int v0){

// 用Dijkstra算法求有向网G的v0顶点到其余顶点v的最短路径P[v]及其带权长度D[v]

// 若P[v][w]为TRUE, 则w是从v0到v当前求得最短路径上的顶点

// final[v]为TRUE当且仅当v属于S, 即已经求得从v0到v的最短路径

int final[G.vexnum]; // final[i]为标志数组, 标记是否已经求得从v0到v的最短路径

int D[G.vexnum]; // D[i]表示当前所找到的从始点v到每个终点vi的最短路径的长度

int P[G.vexnum][G.vexnum]; // P[v][w]见上述第二行注释

int count[G.vexnum]; // count辅助Path的存储

int Path[G.vexnum][G.vexnum]; // Path[i][w]存储v0到vi依次经过哪些点

// printf("\n%d", G.vexnum);

// int i, j;

int v;

for (v=0; v<G.vexnum; ++v){

final[v] = FALSE; D[v] = G.arcs[v0][v].adj; // printf("\n%d", D[i]);

for (int w=0; w<G.vexnum; ++w) {P[v][w] = FALSE, Path[v][w] = FALSE; count[w] = 0;}; // 设空路径

if (D[v] < INFINITY) {P[v][v0] = TRUE; P[v][v] = TRUE; Path[v][v0] = v;}

}// for

// 检验D矩阵

// printf("D如下:\n");

// for (int i=0; i<G.vexnum; ++i){

// printf("%d ", D[i]);

// }

//

// 检验P矩阵

// printf("\nP如下: \n");

// for (int i=1; i<G.vexnum; ++i){

// for (int j=0; j<G.vexnum; ++j){

// printf("%d ", P[i][j]);

// }

// printf("\n");

// }

// 错误的算法尝试

// int MAX = 0; // MAX存储图中的权值最大值, 除无穷

// for (int i=0; i<G.vexnum; ++i){

// for (int j=0; j<G.vexnum; ++j){

// if(G.arcs[i][j].adj != INFINITY && G.arcs[i][j].adj > MAX) MAX = G.arcs[i][j].adj;

// }// for

// }// for

//

// // 将图中权值为无穷的重新赋值(加1), 防止后期算法溢出导致D[i]计算错误

// for (int i=0; i<G.vexnum; ++i){

// for (int j=0; j<G.vexnum; ++j){

// if (G.arcs[i][j].adj == INFINITY) G.arcs[i][j].adj = MAX+1;

// }// for

// }// for

D[v0] = 0; final[v0] = TRUE; // 初始化, v0顶点属于S集合

// 开始主循环, 每次求得v0到某个v顶点的最短路径, 并加v到S集合

int min;

for (int i=1; i<G.vexnum; ++i){ // 其余G.vexnum-1个顶点

min = INFINITY; // 当前所知离v0顶点的最近距离

for (int w=0; w<G.vexnum; ++w){

if(!final[w]) // w顶点在V-S中

if (D[w] < min) {v = w; min = D[w];} // w顶点离v0顶点更近

}// for

final[v] = TRUE; // 离v0顶点最近的v加入S集合

for (int w=0; w<G.vexnum; ++w){ // 更新当前最短路径及距离

if (!final[w] && (min+G.arcs[v][w].adj)<D[w] && G.arcs[v][w].adj<INFINITY && min<INFINITY){ // 修改D[w]和P[w], w属于V-S

// if (!final[w] && (min+G.arcs[v][w].adj)<D[w]){ // 修改D[w]和P[w], w属于V-S

// 上条条件语句判断中当min与无穷相加时会溢出, 得出负的很小的值, 是循环条件无意义

// 因此在前引入MAX变量, 并令图中每个等于无穷的权值均为MAX+1

// 但此操作修改了图的值, 不具有普遍意义, 故放弃

D[w] = min + G.arcs[v][w].adj; count[w] = count[v];

for (int j=0; j<G.vexnum; ++j){ // 更新标志数组和路径存储的数组, 方便后期打印

P[w][j] = P[v][j]; P[w][w] = TRUE;

Path[w][j] = Path[v][j];

}// for

count[w]++; // 辅助数组运作, 使得Path依次存储经过了哪些点

Path[w][count[w]] = w;

}//if

}// for

}// for

for (int i=0; i<G.vexnum; ++i){

// if (D[i] == INFINITY || D[i]>MAX) D[i] = 0; 改进方法, 已弃用

if (D[i] == INFINITY) D[i] = 0;

}// for

// 检验Path矩阵

// printf("\n");

// for(int i=0; i<G.vexnum; ++i){

// for (int j=0; j<G.vexnum; ++j){

// printf("%d ", Path[i][j]);

// }

// printf("\n");

// }

printf("现打印v%d到各点的最短路径及权值:\n", v0);

// for (int i=0; i<G.vexnum; ++i) printf("%d ", D[i]);

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

if (D[i] != 0){ // D[i]等于0则v0到vi没有路径

printf("\nv%d到v%d点之间的最短路径为: %d.\n",v0, i, D[i]);

printf("这之间经过的点依次为: ");

printf("v%d", v0);

for (int j=0; j<G.vexnum; ++j){

if (Path[i][j]!=0) printf("->v%d", Path[i][j]); // Path[i][w]存储v0到vi依次经过哪些点

}// for

printf("\n");

}// if

else printf("v%d到v%d点之间没有最短路径.\n",v0, i);

}// for

}// ShortestPath\_DIJ

int main()

{

// 创建图, 并用邻接矩阵表示

MGraph G;

printf("请输入你想要构建的图的种类:\n");

printf("其中, 1代表有向图、2代表有向网、3代表无向图、4代表无向网\n");

CreateGraph(&G);

// 深度优先搜索图

printf("\n1. 现对图从0号顶点进行深度优先搜索:\n");

DFSTraverse(G);

printf("\n");

// 广度优先搜索图

printf("\n2. 现对图从0号顶点进行广度优先搜索:\n");

BFSTraverse(G);

printf("\n");

// 求得最短路径

printf("\n3. 现用Dijskrta求解最短路径问题.\n");

printf("请输入你选择的起始顶点的序号: ");

int v0;

scanf("%d", &v0);

ShortestPath\_DIJ(G, v0);

printf("\nOK");

}// main

1. 成果截图:

