哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：数据结构与算法课程

类型：必修

实验项目：图型结构的建立与搜索

实验题目： 图的存储结构的建立与搜索

实验日期： 2018.12.06

班级：1703002

学号：1170500913

姓名：熊健羽

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 张岩 |

1. **实验目的**
2. **实验要求及实验环境**
   1. 实验要求
   2. 实验环境
      1. 硬件环境

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64位)

GPU：Intel(R) HD Graphics 620

Nvidia GeForce 940MX

物理内存：8.00GB

磁盘：1TB HDD

128GB SSD

* + 1. 软件环境：

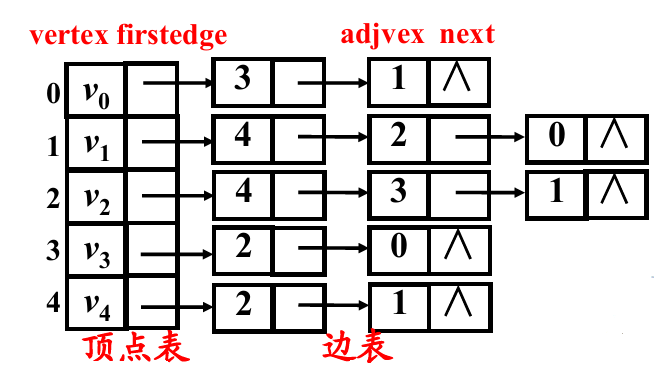
OS: Microsoft Windows 10 家庭中文版

编译环境：MinGW-W64

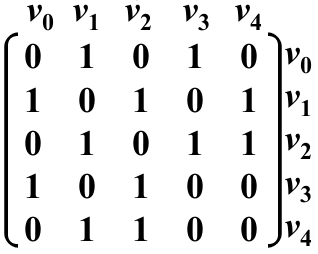
IDE：VS code、code::blocks

1. **设计思想**（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系）
   1. 逻辑设计
      1. 数据结构:

领接表：



邻接矩阵：



* + 1. 各个函数的功能及其调用关系：
       1. 函数原型：UDgraph\_ED makeNull\_ED(void)

函数功能：初始化一个空的邻接矩阵图

函数调用：malloc

* + - 1. 函数原型：UDgraph\_ADJ makeNull\_ADJ(void)

函数功能：初始化一个空的邻接表图

函数调用：malloc

* + - 1. 函数原型：bool isEdgeRight(UDgraph\_ED G)

函数功能：判断输入的边是否正确

函数调用：无

* + - 1. 函数原型：UDgraph\_ED build\_ED(char \*path)

函数功能：构建以邻接矩阵存储的图

函数调用：makeNull\_ED，fopen, fgets, sscanf,printf,exit

* + - 1. 函数原型：UDgraph\_ADJ build\_ADJ(char \*path)

函数功能：构建以邻接表存储的图

函数调用：makeNull\_ADJ，fopen, fgets, sscanf,printf,exit

* + - 1. 函数原型：void printEdge(UDgraph\_ED G)

函数功能：打印图的邻接矩阵

函数调用：printf

* + - 1. 函数原型：ED2ADJ(UDgraph\_ADJ G\_adj, UDgraph\_ED G\_ed)

函数功能：邻接矩阵形式转为邻接表形式

函数调用：无

* + - 1. 函数原型：ADJ2ED(UDgraph\_ED G\_ed, UDgraph\_ADJ G\_adj)

函数功能：邻接矩阵形式转为邻接矩阵形式

函数调用：无

* + - 1. 函数原型：void DFS\_ED\_Recur(UDgraph\_ED G, int start, bool visited[], int dfn[], int \*counter)

函数功能：以某个节点开始递归先深（邻接矩阵）

函数调用：DFS\_ED\_Recur

* + - 1. 函数原型：int \*DFSTraverse\_ADJ\_Recur(UDgraph\_ADJ G)

函数功能：先深递归主算法（邻接矩阵）

函数调用：DFS\_ED\_Recur

* + - 1. 函数原型：void DFS\_ADJ\_Recur(UDgraph\_ADJ G, int start, bool visited[], int dfn[], int \*counter)

函数功能：从某个节点开始先深，递归，（邻接表）

函数调用：DFS\_ADJ\_Recur

* + - 1. 函数原型：int \*DFSTraverse\_ADJ\_Recur(UDgraph\_ADJ G)

函数功能：先深递归主算法（邻接表）

函数调用：DFS\_ADJ\_Recur

* + - 1. 函数原型：int FirstNextIndex\_ED(UDgraph\_ED G, int i, bool \*visited, STACK \*st)

函数功能：返回第一个邻接的节点号，其他邻接节点入栈

函数调用：无

* + - 1. 函数原型：void DFS\_ED\_NoRecur(UDgraph\_ED G, int i, bool \*visited, int dfn[], int \*counter)

函数功能：从某节点开始，非递归先深（邻接矩阵）

函数调用：FirstNextIndex\_ED，printf

* + - 1. 函数原型：int \*DFS\_ADJ\_NoRecur(UDgraph\_ADJ G, int i, bool \*visited, int dfn[], int \*counter)

函数功能： 非递归先深主算法（邻接表）

函数调用：DFS\_ADJ\_NoRecur

* + - 1. 函数原型：void BFS\_ED(UDgraph\_ED G, int i, bool \*visited, int \*bfn, int \*counter)

函数功能：从某节点开始先广（邻接矩阵）

函数调用：printf

* + - 1. 函数原型：int \*BFSTraverse\_ED(UDgraph\_ED G)

函数功能：先广主算法（邻接矩阵）

函数调用：BFS\_ED

* + - 1. 函数原型：void BFS\_ADJ(UDgraph\_ADJ G, int i, bool \*visited, int \*bfn, int \*counter)

函数功能：从某点开始先广（邻接表）

函数调用：printf

* + - 1. 函数原型：int \*BFSTraverse\_ADJ(UDgraph\_ADJ G)

函数功能：先广主算法（邻接表）

函数调用：BFS\_ADJ

* + - 1. 函数原型：void printTree(Tree t, int num)

函数功能： 打印先深（广）树

函数调用：printf

* + - 1. 函数原型：void printDfnBfn(int \*fn, int num)

函数功能： 打印深度\广度优先树

函数调用：printf

* 1. 物理设计
     1. 重要函数的实现思路：
        1. 函数原型：bool isEdgeRight(UDgraph\_ED G)

函数功能：判断输入的边是否正确

函数思路：判断对角线是否为零，是否为对称矩阵

* + - 1. 函数原型：UDgraph\_ED build\_ED(char \*path)

函数功能：构建以邻接矩阵存储的图

函数思路：分别读入文件的每一行，识别并获取节点数、节点值、边数、邻接矩阵等信息，并存入存储结构中

* + - 1. 函数原型：UDgraph\_ADJ build\_ADJ(char \*path)

函数功能：构建以邻接表存储的图

函数思路：分别读入文件的每一行，识别并获取节点数、节点值、边数、边等信息，并存入存储结构中

* + - 1. 函数原型：ED2ADJ(UDgraph\_ADJ G\_adj, UDgraph\_ED G\_ed)

函数功能：邻接矩阵形式转为邻接表形式

函数思路：遍历邻接矩阵的每一行，遇到1，则在相应的顶点表后面添加相应的边表

* + - 1. 函数原型：ADJ2ED(UDgraph\_ED G\_ed, UDgraph\_ADJ G\_adj)

函数功能：邻接矩阵形式转为邻接矩阵形式

函数思路：遍历邻接表的每个顶点表后面的边表，把相应行相应列的邻接矩阵置为1

* + - 1. 函数原型：void DFS\_ED\_Recur(UDgraph\_ED G, int start, bool visited[], int dfn[], int \*counter)

函数功能：以某个节点开始递归先深（邻接矩阵）

函数思路：

0.所有顶点标记为未访问visited[v]={0,…};

1. 访问顶点v; visited[v]=1;

2. w=顶点v的第一个邻接点；

3. while (w存在)

3.1 if (w未被访问)

从顶点w出发递归地执行该算法;

3.2 w=顶点v的下一个邻接点;

* + - 1. 函数原型：void DFS\_ADJ\_Recur(UDgraph\_ADJ G, int start, bool visited[], int dfn[], int \*counter)

函数功能：从某个节点开始先深，递归，（邻接表）

函数思路：DFS\_ADJ\_Recur

* + - 1. 函数原型：int FirstNextIndex\_ED(UDgraph\_ED G, int i, bool \*visited, STACK \*st)

函数功能：返回i号节点第一个邻接的节点号，其他邻接节点入栈

函数思路：设置flag，表示是否找到了第一个邻接的节点号。遍历第i行邻接矩阵，结合visited数组，找到了则记录到ret里。下来，多余的入栈。最后再返回。

* + - 1. 函数原型：void DFS\_ED\_NoRecur(UDgraph\_ED G, int i, bool \*visited, int dfn[], int \*counter)

函数功能：从某节点开始，非递归先深（邻接矩阵）

函数思路：维护一个栈，存储节点序号。每次访问一个节点，获取下一个节点的同时，其他邻接的节点入栈，如此直到尽头。则出栈，再重复上述过程。直到栈空，且无邻接且未访问过的节点，则遍历结束

* + - 1. 函数原型：void BFS\_ED(UDgraph\_ED G, int i, bool \*visited, int \*bfn, int \*counter)

函数功能：从某节点开始先广（邻接矩阵）

函数思路：维护一个队列。

1 . 初始化队列Q;

2. 访问顶点v; visited [v]=1; 顶点v入队Q;

3. while (队列Q非空)

3.1 v=队列Q的队头元素出队;

3.2 w=顶点v的第一个邻接点;

3.3 while (w存在) //访问v的所有邻接点

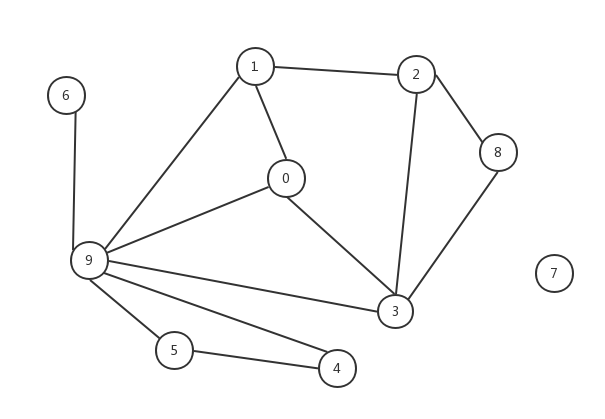
3.3.1 如果w 未被访问，则

访问顶点w; visited[w]=1; 顶点w入队列Q；

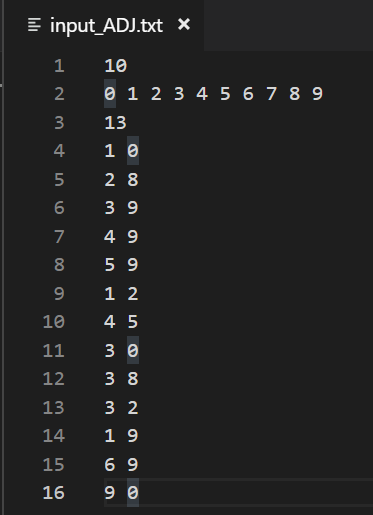
3.3.2 w=顶点v的下一个邻接点；

1. **测试结果**

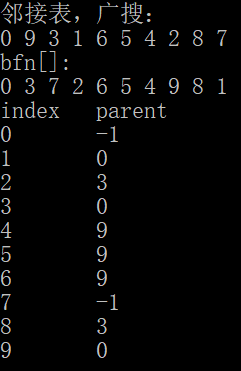
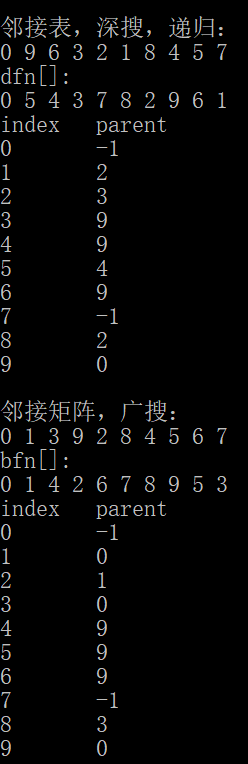
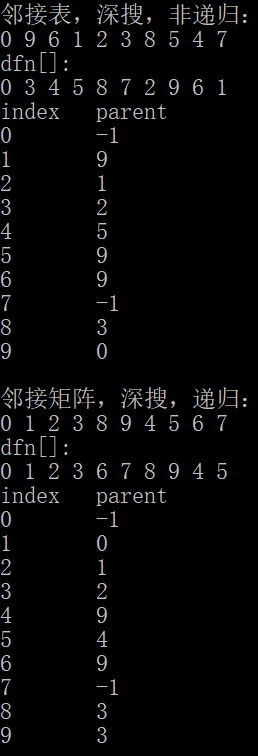
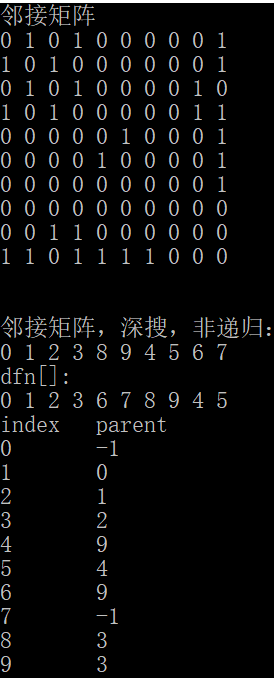
构建如图所示的图：



输入的文件如下：



运行结果如下：



1. **经验体会与不足**

经验体会：

两种存储结构各有优劣，邻接表适合边稀疏图，邻接矩阵适合边稠密图。

同一个图，使用两种存储结构存储，其遍历的顺序也会有所不同。

同一个图的同一种存储结构，使用递归和非递归的遍历，遍历顺序也可能不同（如邻接表存储的图，非递归时，入栈顺序是顺着边表指针方向入栈，也就是逆着边表指针方向的次序出栈；但递归时则不一样，永远是顺着边表指针方向依次递归访问的。考虑到算法的简洁，两者难免会有区别）。

不足：

树形结构只使用父链表示，且只表示了树边，未表示出横边（回退边）。

能力所限，无法生成图的图形化表示。

1. **附录：源代码（带注释）**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <stdbool.h>

#define MAX 10000

**typedef** **struct**

{

**int** parent[MAX];

**int** n;

} Tree; *//先深（广）生成树*

Tree t1, t2, t3, t4, t5, t6;

**typedef** **int** datatype;

**typedef** **struct**

{

datatype vertex[MAX];

bool edge[MAX][MAX];

**int** n, e;

} UDgraph\_ED\_NODE; *//邻接矩阵表示的图*

**typedef** **struct** EDGENODE

{

**int** adjvex;

**struct** EDGENODE \*next;

} EdgeNode; *//邻接表的边表节点*

**typedef** **struct**

{

datatype vertex;

EdgeNode \*firstedge;

} VertexNode; *//邻接表的顶点表节点*

**typedef** **struct**

{

VertexNode vexlist[MAX];

**int** n, e;

} UDgraph\_ADJ\_NODE; *//邻接表表示的图*

**typedef** UDgraph\_ADJ\_NODE \*UDgraph\_ADJ;

**typedef** UDgraph\_ED\_NODE \*UDgraph\_ED;

UDgraph\_ED makeNull\_ED(**void**) *//初始化一个空的邻接矩阵图*

{

UDgraph\_ED G = (UDgraph\_ED)malloc(**sizeof**(UDgraph\_ED\_NODE));

**for** (**int** i = 0; i < MAX; i++)

**for** (**int** j = 0; j < MAX; j++)

G->edge[i][j] = 0;

G->e = 0;

G->n = 0;

**return** G;

}

UDgraph\_ADJ makeNull\_ADJ(**void**) *//初始化一个空的邻接表图*

{

UDgraph\_ADJ G = (UDgraph\_ADJ)malloc(**sizeof**(UDgraph\_ADJ\_NODE));

G->n = 0;

G->e = 0;

**for** (**int** i = 0; i < MAX; i++)

G->vexlist[i].firstedge = NULL;

**return** G;

}

bool isEdgeRight(UDgraph\_ED G) *//判断输入的边是否正确*

{

**int** e\_count = 0;

**for** (**int** i = 0; i < G->n; i++)

{

**for** (**int** j = 0; j < G->n; j++)

{

**if** (G->edge[i][j])

e\_count++;

**if** (G->edge[i][j] != G->edge[j][i])

**return** false;

**if** (i == j && G->edge[i][j] == 1)

**return** false;

}

}

**if** (e\_count / 2 == G->e)

**return** true;

**else**

**return** false;

}

*// void readNumbers(int n, int)*

UDgraph\_ED build\_ED(**char** \*path) *//构建邻接矩阵图*

{

UDgraph\_ED G = makeNull\_ED();

**FILE** \*fp;

**if** ((fp = fopen(path, "r+")) == NULL)

{

printf("open failed!**\n**");

exit(1);

}

**char** s[MAX];

**int** line = 0;

**int** n\_input, e\_input;

**char** \*f\_ch;

**int** node\_num = 0;

**int** counter;

**int** row = 0, column;

**while** (fgets(s, MAX, fp) != NULL)

{

line++; *//add line*

**switch** (line)

{

**case** 1: *//line 1*

sscanf(s, "%d", &n\_input);

G->n = n\_input;

**break**;

**case** 2: *//line 2*

f\_ch = s;

counter = 0;

**while** (\*f\_ch)

{

**if** (\*f\_ch >= '0' && \*f\_ch <= '9') *//if get number*

{

sscanf(f\_ch, "%d", &(G->vertex[counter]));

counter++;

**while** (\*f\_ch >= '0' && \*f\_ch <= '9') *//jump out of number*

f\_ch++;

f\_ch--;

}

f\_ch++;

}

**if** (counter != n\_input) *//if num is mismatching,print warning*

printf("warning: there is node which is not intialized!**\n**");

**break**;

**case** 3: *//line 3*

sscanf(s, "%d", &e\_input);

G->e = e\_input;

**break**;

**default**: *//input edge*

f\_ch = s;

column = 0;

**while** (\*f\_ch)

{

**if** (\*f\_ch == '1')

G->edge[row][column++] = 1;

**else** **if** (\*f\_ch == '0')

G->edge[row][column++] = 0;

f\_ch++;

}

**if** (column != n\_input)

{

printf("the edge input error!**\n**");

exit(2);

}

row++;

}

}

**if** (!isEdgeRight(G)) *//判断边的合法性*

{

printf("the the edge input error!**\n**");

exit(3);

}

fclose(fp);

**return** G;

}

UDgraph\_ADJ build\_ADJ(**char** \*path) *//构建邻接表图*

{

UDgraph\_ADJ G = makeNull\_ADJ();

**FILE** \*fp;

**if** ((fp = fopen(path, "r+")) == NULL)

{

printf("open failed!**\n**");

exit(1);

}

**char** s[MAX];

**char** \*f\_ch;

**int** line = 0, n\_input, e\_input;

**int** e\_counter = 0;

**int** n1, n2;

**while** (fgets(s, MAX, fp) != NULL)

{

line++;

**switch** (line)

{

**case** 1:

sscanf(s, "%d", &n\_input);

**break**;

**case** 2:

f\_ch = s;

**int** counter = 0;

**while** (\*f\_ch)

{

**if** (\*f\_ch >= '0' && \*f\_ch <= '9')

{

sscanf(f\_ch, "%d", &(G->vexlist[counter]));

counter++;

**while** (\*f\_ch >= '0' && \*f\_ch <= '9')

f\_ch++;

f\_ch--;

}

f\_ch++;

}

**if** (counter != n\_input)

printf("warning: There is vexlist which is not initialized!**\n**");

**break**;

**case** 3:

sscanf(s, "%d", &e\_input);

**break**;

**default**:

**if** (sscanf(s, "%d %d", &n1, &n2) != 2 || n1 < 0 || n1 >= n\_input || n2 < 0 || n2 >= n\_input)

{

printf("input error!**\n**");

exit(2);

}

**int** flag = 0;

EdgeNode \*p = G->vexlist[n1].firstedge;

**while** (p)

{

**if** (p->adjvex == n2)

flag = 1;

p = p->next;

}

**if** (!flag && n1 != n2)

{

p = (EdgeNode \*)malloc(**sizeof**(EdgeNode));

p->adjvex = n2;

p->next = G->vexlist[n1].firstedge;

G->vexlist[n1].firstedge = p;

p = (EdgeNode \*)malloc(**sizeof**(EdgeNode));

p->adjvex = n1;

p->next = G->vexlist[n2].firstedge;

G->vexlist[n2].firstedge = p;

e\_counter++;

}

}

}

**if** (e\_counter != e\_input)

{

printf("the number of edge is error!**\n**");

exit(2);

}

G->n = n\_input;

G->e = e\_input;

**return** G;

}

**void** printEdge(UDgraph\_ED G) *//打印图的邻接表表示*

{

**for** (**int** i = 0; i < G->n; i++)

{

**for** (**int** j = 0; j < G->n; j++)

printf("%d ", G->edge[i][j]);

printf("**\n**");

}

}

**void** ED2ADJ(UDgraph\_ADJ G\_adj, UDgraph\_ED G\_ed) *//邻接矩阵形式转为邻接表形式*

{

**int** i, j;

EdgeNode \*p\_edgeNode;

G\_adj->n = G\_ed->n;

G\_adj->e = G\_ed->e;

**for** (i = 0; i < G\_ed->n; i++)

G\_adj->vexlist[i].vertex = G\_ed->vertex[i];

**for** (i = 0; i < G\_ed->n; i++)

{

**for** (j = 0; j < G\_ed->n; j++)

{

**if** (G\_ed->edge[i][j])

{

p\_edgeNode = (EdgeNode \*)malloc(**sizeof**(EdgeNode));

p\_edgeNode->adjvex = j;

p\_edgeNode->next = G\_adj->vexlist[i].firstedge;

G\_adj->vexlist[i].firstedge = p\_edgeNode;

}

}

}

}

**void** ADJ2ED(UDgraph\_ED G\_ed, UDgraph\_ADJ G\_adj) *//邻接矩阵形式转为邻接矩阵形式*

{

**int** i, j;

EdgeNode \*p\_edgeNode;

G\_ed->n = G\_adj->n;

G\_ed->e = G\_adj->e;

**for** (i = 0; i < G\_adj->n; i++)

G\_ed->vertex[i] = G\_adj->vexlist[i].vertex;

**for** (i = 0; i < G\_adj->n; i++)

{

p\_edgeNode = G\_adj->vexlist[i].firstedge;

**while** (p\_edgeNode)

{

G\_ed->edge[i][p\_edgeNode->adjvex] = 1;

p\_edgeNode = p\_edgeNode->next;

}

}

}

**void** DFS\_ED\_Recur(UDgraph\_ED G, **int** start, bool visited[], **int** dfn[], **int** \*counter) *//以某个节点开始先深，递归（邻接矩阵）*

{

visited[start] = true;

printf("%d ", G->vertex[start]); *//输出节点*

dfn[start] = \*counter; *//更新先深序列*

(\*counter)++; *//计数器加一*

**for** (**int** i = 0; i < G->n; i++)

{

**if** (G->edge[start][i] == 1 && visited[i] == false) *//访问下一个邻接的，未被访问的节点*

{

DFS\_ED\_Recur(G, i, visited, dfn, counter);

t1.parent[i] = start; *//构造先深生成树*

}

}

}

**int** \*DFSTraverse\_ED\_Recur(UDgraph\_ED G) *//先深递归（邻接矩阵）*

{

**int** \*dfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

**int** counter = 0;

bool visited[MAX];

**for** (**int** i = 0; i < MAX; i++) *//初始化先深生成树*

t1.parent[i] = -1;

**for** (**int** i = 0; i < G->n; i++)

visited[i] = false; *//初始化访问数组*

**for** (**int** i = 0; i < G->n; i++) *//从每个未被访问的节点开始*

**if** (!visited[i])

DFS\_ED\_Recur(G, i, visited, dfn, &counter);

**return** dfn;

}

**void** DFS\_ADJ\_Recur(UDgraph\_ADJ G, **int** start, bool visited[], **int** dfn[], **int** \*counter)

*//从某个节点开始先深，递归，（邻接表） （与上同理）*

{

visited[start] = true;

printf("%d ", G->vexlist[start].vertex); *//输出节点*

dfn[start] = \*counter; *//更新先深序列*

(\*counter)++; *//计数器加一*

**for** (EdgeNode \*p = G->vexlist[start].firstedge; p != NULL; p = p->next) *//遍历所有边表节点*

{

**if** (visited[p->adjvex] == false) *//若没访问*

{

t2.parent[p->adjvex] = start;

DFS\_ADJ\_Recur(G, p->adjvex, visited, dfn, counter);

}

}

}

**int** \*DFSTraverse\_ADJ\_Recur(UDgraph\_ADJ G) *// 先深，递归，（邻接表） （与上同理）*

{

**int** \*dfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

bool visited[MAX];

**int** counter = 0;

**for** (**int** i = 0; i < MAX; i++)

t2.parent[i] = -1;

**for** (**int** i = 0; i < MAX; i++)

visited[i] = false;

**for** (**int** i = 0; i < G->n; i++)

{

**if** (!visited[i])

DFS\_ADJ\_Recur(G, i, visited, dfn, &counter);

}

**return** dfn;

}

**typedef** **struct** *//非递归用到的栈*

{

**int** data[MAX];

**int** top;

} STACK;

**int** FirstNextIndex\_ED(UDgraph\_ED G, **int** i, bool \*visited, STACK \*st) */\*返回第一个邻接的节点号*

*其他邻接节点入栈\*/*

{

**int** ret, flag = 0; *//flag表示是否找到未访问的第一个节点*

**for** (**int** j = 0; j < G->n; j++)

{

**if** (G->edge[i][j] && visited[j] == false)

{

t3.parent[j] = i; *//更新先深生成树*

**if** (!flag)

{

ret = j;

flag = 1;

}

**else**

{

st->data[++(st->top)] = j;

}

}

}

**if** (flag)

**return** ret;

**else**

**return** -1; *//未找到，返回 -1*

}

**void** DFS\_ED\_NoRecur(UDgraph\_ED G, **int** i, bool \*visited, **int** dfn[], **int** \*counter)

*//从某节点开始，非递归先深（邻接矩阵）*

{

STACK \*st = (STACK \*)malloc(**sizeof**(STACK));

st->top = -1;

**int** temp;

visited[i] = true;

printf("%d ", G->vertex[i]);

dfn[i] = (\*counter)++;

i = FirstNextIndex\_ED(G, i, visited, st);

**while** (st->top != -1 || i != -1) *//当栈空且访问到尽头，结束循环*

{

**while** (i != -1) *//循环直到尽头*

{

visited[i] = true;

printf("%d ", G->vertex[i]);

dfn[i] = (\*counter)++;

i = FirstNextIndex\_ED(G, i, visited, st);

}

**if** (st->top != -1) *//如果栈不空*

{

**do**

temp = st->data[(st->top)--];

**while** (visited[temp] && st->top != -1); *//元素出栈，直到出栈的是未访问的*

**if** (!visited[temp]) *//若出栈的是未访问的,则有效*

i = temp;

}

}

}

**int** \*DFSTraverse\_ED\_NoRecur(UDgraph\_ED G) *//非递归先深（邻接矩阵），细节同上，略*

{

**int** \*dfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

**int** counter = 0;

bool visited[MAX];

**for** (**int** i = 0; i < MAX; i++)

t3.parent[i] = -1;

**for** (**int** i = 0; i < G->n; i++)

visited[i] = false;

**for** (**int** i = 0; i < G->n; i++)

**if** (!visited[i])

DFS\_ED\_NoRecur(G, i, visited, dfn, &counter);

**return** dfn;

}

**int** FirstNextIndex\_ADJ(UDgraph\_ADJ G, **int** i, bool \*visited, STACK \*st, **int** \*dfn, **int** \*counter)

*/\*返回第一个邻接的节点号，*

*并访问。*

*其他邻接节点入栈 (与上面同理)\*/*

{

EdgeNode \*p = G->vexlist[i].firstedge;

**int** ret;

**int** flag = 0;

printf("%d ", G->vexlist[i].vertex);

visited[i] = true;

dfn[i] = \*counter;

(\*counter)++;

**while** (p)

{

**if** (!visited[p->adjvex])

{

t4.parent[p->adjvex] = i;

**if** (!flag)

{

flag = 1;

ret = p->adjvex;

}

**else**

{

st->data[++(st->top)] = p->adjvex;

}

}

p = p->next;

}

**if** (flag)

**return** ret;

**else**

**return** -1;

}

**int** \*DFS\_ADJ\_NoRecur(UDgraph\_ADJ G, **int** i, bool \*visited, **int** dfn[], **int** \*counter)

*//从某节点开始，非递归先深（邻接表）*

{

STACK \*st = (STACK \*)malloc(**sizeof**(STACK));

st->top = -1;

**int** temp;

**while** (st->top != -1 || i != -1)

{

**while** (i != -1)

i = FirstNextIndex\_ADJ(G, i, visited, st, dfn, counter);

**if** (st->top != -1)

{

**do**

temp = st->data[(st->top)--];

**while** (visited[temp] && st->top != -1);

**if** (!visited[temp])

i = temp;

}

}

}

**int** \*DFSTraverse\_ADJ\_NoRecur(UDgraph\_ADJ G) *//非递归先深（邻接表）*

{

**int** \*dfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

**int** counter = 0;

bool visited[MAX];

**for** (**int** i = 0; i < MAX; i++)

t4.parent[i] = -1;

**for** (**int** i = 0; i < G->n; i++)

visited[i] = false;

**for** (**int** i = 0; i < G->n; i++)

**if** (!visited[i])

DFS\_ADJ\_NoRecur(G, i, visited, dfn, &counter);

**return** dfn;

}

**void** BFS\_ED(UDgraph\_ED G, **int** i, bool \*visited, **int** \*bfn, **int** \*counter) *//从某节点开始先广（邻接矩阵）*

{

**int** Q[MAX];

**int** front = 0, rear = 0; *//初始化队列*

Q[rear++] = i; *//首节点入队并输出*

bfn[i] = \*counter;

(\*counter)++;

visited[i] = true;

printf("%d ", G->vertex[i]);

**while** (front != rear) *//循环到队空*

{

i = Q[front++]; *//出队*

**for** (**int** j = 0; j < G->n; j++)

{

**if** (G->edge[i][j] && visited[j] == false) *//若邻接且未访问，则入队并输出*

{

t5.parent[j] = i;

Q[rear++] = j;

bfn[j] = \*counter;

(\*counter)++;

visited[j] = true;

printf("%d ", G->vertex[j]);

}

}

}

}

**int** \*BFSTraverse\_ED(UDgraph\_ED G) *//先广（邻接矩阵）*

{

**int** \*bfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

;

**int** counter = 0;

bool visited[MAX];

**for** (**int** i = 0; i < MAX; i++)

t5.parent[i] = -1;

**for** (**int** i = 0; i < G->n; i++)

visited[i] = false;

**for** (**int** i = 0; i < G->n; i++)

**if** (!visited[i])

BFS\_ED(G, i, visited, bfn, &counter);

**return** bfn;

}

**void** BFS\_ADJ(UDgraph\_ADJ G, **int** i, bool \*visited, **int** \*bfn, **int** \*counter)*//从某点开始先广（邻接表）（原理同上）*

{

**int** Q[MAX];

**int** rear = 0, front = 0;

EdgeNode \*p;

Q[rear++] = i;

printf("%d ", G->vexlist[i].vertex);

visited[i] = true;

bfn[i] = \*counter;

(\*counter)++;

**while** (rear != front)

{

i = Q[front++];

p = G->vexlist[i].firstedge;

**while** (p)

{

**if** (!visited[p->adjvex])

{

t6.parent[p->adjvex] = i;

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

printf("%d ", G->vexlist[p->adjvex].vertex);

visited[p->adjvex] = true;

bfn[p->adjvex] = \*counter;

(\*counter)++;

}

p = p->next;

}

}

}

**int** \*BFSTraverse\_ADJ(UDgraph\_ADJ G) *//先广（邻接表）*

{

**int** \*bfn = (**int** \*)calloc(G->n, **sizeof**(**int**));

bool visited[MAX];

**int** counter = 0;

**for** (**int** i = 0; i < MAX; i++)

t6.parent[i] = -1;

**for** (**int** i = 0; i < G->n; i++)

visited[i] = false;

**for** (**int** i = 0; i < G->n; i++)

**if** (!visited[i])

BFS\_ADJ(G, i, visited, bfn, &counter);

**return** bfn;

}

**void** printTree(Tree t, **int** num) *//打印先深（广）树*

{

printf("index**\t**parent**\n**");

**for** (**int** i = 0; i < num; i++)

{

printf("%d**\t**%d**\n**", i, t.parent[i]);

}

}

**void** printDfnBfn(**int** \*fn, **int** num) *//打印深度\广度优先树*

{

**for** (**int** i = 0; i < num; i++)

printf("%d ", fn[i]);

printf("**\n**");

}

**int** main()

{

UDgraph\_ADJ G\_adj = build\_ADJ(".**\\**input\_ADJ.txt");

UDgraph\_ED G\_ed = makeNull\_ED();

ADJ2ED(G\_ed, G\_adj);

*// UDgraph\_ED G\_ed = build\_ED("F:\\data\_structure\\Lab\\lab3\\input\_ED.txt");*

*// UDgraph\_ADJ G\_adj = makeNull\_ADJ();*

*// ED2ADJ(G\_adj, G\_ed);*

printf("邻接矩阵**\n**");

printEdge(G\_ed);

printf("**\n\n**");

printf("邻接矩阵，深搜，非递归：**\n**");

**int** \*dfn\_ed\_recur = DFSTraverse\_ED\_Recur(G\_ed);

printf("**\n**");

printf("dfn[]:**\n**");

printDfnBfn(dfn\_ed\_recur, G\_ed->n);

printTree(t1, G\_ed->n);

printf("**\n**");

printf("邻接表，深搜，非递归：**\n**");

**int** \*dfn\_adj\_recur = DFSTraverse\_ADJ\_Recur(G\_adj);

printf("**\n**");

printf("dfn[]:**\n**");

printDfnBfn(dfn\_adj\_recur, G\_adj->n);

printTree(t2, G\_adj->n);

printf("**\n**");

printf("邻接矩阵，深搜，递归：**\n**");

**int** \*dfn\_ed\_Norecur = DFSTraverse\_ED\_NoRecur(G\_ed);

printf("**\n**");

printf("dfn[]:**\n**");

printDfnBfn(dfn\_ed\_Norecur, G\_ed->n);

printTree(t3, G\_ed->n);

printf("**\n**");

printf("邻接表，深搜，递归：**\n**");

**int** \*dfn\_adj\_Norecur = DFSTraverse\_ADJ\_NoRecur(G\_adj);

printf("**\n**");

printf("dfn[]:**\n**");

printDfnBfn(dfn\_adj\_Norecur, G\_adj->n);

printTree(t4, G\_adj->n);

printf("**\n**");

printf("邻接矩阵，广搜：**\n**");

**int** \*bfn\_ed = BFSTraverse\_ED(G\_ed);

printf("**\n**");

printf("bfn[]:**\n**");

printDfnBfn(bfn\_ed, G\_ed->n);

printTree(t5, G\_ed->n);

printf("**\n**");

printf("邻接表，广搜：**\n**");

**int** \*bfn\_adj = BFSTraverse\_ADJ(G\_adj);

printf("**\n**");

printf("bfn[]:**\n**");

printDfnBfn(bfn\_adj, G\_adj->n);

printTree(t6, G\_adj->n);

printf("**\n**");

getchar();

**return** 0;

}