****实验5：图的深度和广度有优先遍历（学时：6）****

实验内容：建立一个图，对图进行深度优先遍历及广度优先遍历。

基本内容：只对无向图进行深度和广度优先遍历。

选做内容：

计算图中两个顶点之间的最短路径，如果不存在路径，则提示没有路径。

算法描述：

1. 创建邻接表和图（尾插法确保邻接表的顺序）
2. 深度优先遍历（递归），取邻接表的下一个未访问节点
3. 广度优先遍历，遍历一个节点的链表后在遍历。
4. 最小路径dijkstra算法，写出寻找最小路径算法，在更新最小路径。

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_VERTICES 100

// 图中的节点

typedef struct Node {

int vertex; // 节点编号

int weight; // 距离/权重

struct Node\* next; // 指向下一个节点的指针

} Node;

// 图

typedef struct Graph {

Node\* adjList[MAX\_VERTICES]; // 邻接表数组

int numVertices; // 图中节点的数量

} Graph;

// 创建节点

Node\* createNode(int vertex, int weight) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

// 创建图

Graph\* createGraph(int numVertices) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->numVertices = numVertices;

// 初始化邻接表数组为空

int i;

for (i = 1; i <= numVertices; i++) {

graph->adjList[i] = NULL;

}

return graph;

}

// 添加边到图中

void addEdge(Graph\* graph, int src, int dest, int weight) {

// 添加从源节点到目标节点的边

Node\* newNode = createNode(dest, weight);

if (graph->adjList[src] == NULL) {

graph->adjList[src] = newNode;

}

else {

Node\* temp = graph->adjList[src];

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// 添加从目标节点到源节点的边（因为是无向图）

newNode = createNode(src, weight);

if (graph->adjList[dest] == NULL) {

graph->adjList[dest] = newNode;

}

else {

Node\* temp = graph->adjList[dest];

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

//深度优先遍历和广度优先遍历

void Dfs(Graph\* graph,int vertex,int visit[]) {

visit[vertex] = 1;

printf("%d", vertex);

Node\* temp = graph->adjList[vertex];

while (temp != NULL) {

int adjVertex = temp->vertex;

if (!visit[adjVertex]) {

Dfs(graph, adjVertex,visit);

}

temp = temp->next;

}

}

void Bfs(Graph\* graph, int vertex,int visit[]) {

visit[vertex] = 1;

int queue[MAX\_VERTICES];//队列，存要访问的节点

int front = 0, rear = 0;//头尾

queue[rear++] = vertex;//起始节点入队

while (front < rear) {

int cVertex = queue[front++];//取一个节点

printf("%d", cVertex);

Node\* temp = graph->adjList[cVertex];//遍历当前节点的链表

while (temp != NULL) {

int adjVertex = temp->vertex;

if (!visit[adjVertex]) {

visit[adjVertex] = 1;

queue[rear++] = adjVertex;//入队

}

temp = temp->next;

}

}

}

// 查找最小路径

void findMinimumPath(Graph\* graph, int startVertex, int endVertex) {

int dist[MAX\_VERTICES]; // 存储从起始节点到各个节点的最小距离

int visited[MAX\_VERTICES]; // 记录节点是否被访问

int previous[MAX\_VERTICES]; // 记录路径上的前一个节点,前驱数组

// 初始化数组

int i;

for (i = 1; i <= graph->numVertices; i++) {

dist[i] = 100;//初始化每个值都为很大

visited[i] = 0;

previous[i] = -1;

}

// 设置起始节点的距离为0

dist[startVertex] = 0;

// 找到最小路径

int count;

for (count = 0; count < graph->numVertices - 1; count++) {

int minDist = 100;//初始化路径很大

int minVertex = -1;//初始化前驱为没有前驱（-1代替）

// 选择距离最小且未被访问的节点

for (i = 1; i <= graph->numVertices; i++) {

if (!visited[i] && dist[i] < minDist) {

minDist = dist[i];

minVertex = i;

}

}

// 标记选中的节点为已访问

visited[minVertex] = 1;

// 更新最小路径

Node\* temp = graph->adjList[minVertex];

while (temp != NULL) {

int neighbor = temp->vertex;

int weight = temp->weight;

if (!visited[neighbor] && dist[minVertex] != 100 && dist[minVertex] + weight < dist[neighbor]) {

dist[neighbor] = dist[minVertex] + weight;

previous[neighbor] = minVertex;

}

temp = temp->next;

}

}

if (dist[endVertex] == 100) {

printf("无法到达！");

}

else {

// 输出最小路径

printf("最小路径: ");

int path[MAX\_VERTICES];//存节点路劲的栈

int pathLength = 0;//栈指针

int currentVertex = endVertex;//前驱节点

while (currentVertex != -1) {//从尾到头依次存入

path[pathLength++] = currentVertex;

currentVertex = previous[currentVertex];

}

// 对于栈，反向输出路径；对于人的逻辑，正向输出

for (i = pathLength - 1; i >= 0; i--) {

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

}

printf("\n");

// 输出最小距离

printf("最小距离: %d\n", dist[endVertex]);

}

}

int main() {

printf("请输入节点数目：");

int numVertices;

scanf("%d", &numVertices);

Graph\* graph = createGraph(numVertices);

// 添加边到图中

int src, dest, weight;

int numEdges;

printf("请输入要添加的边的数量：");

scanf("%d", &numEdges);

for (int i = 0; i < numEdges; i++) {

printf("请输入边的起始节点、目标节点和权重");

scanf("%d %d %d", &src, &dest, &weight);

addEdge(graph, src, dest, weight);

}

//深度优先遍历和广度优先遍历

int visit[MAX\_VERTICES] = { 0 };

printf("深度优先遍历结果：");

for (int i = 1; i <= graph->numVertices; i++) {

if (!visit[i]) {

Dfs(graph, i, visit);

}

}

for (int i = 1; i <= graph->numVertices; i ++) {

visit[i] = 0;

}

printf("广度优先遍历结果：");

for (int i = 1; i <= graph->numVertices; i++) {

if (!visit[i]) {

Bfs(graph, i, visit);

printf("\n");

}

}

printf("请输入最小路径的起点和终点：");

int startVertex;

int endVertex;

scanf("%d %d", &startVertex, &endVertex);

// 查找最小路径

findMinimumPath(graph, startVertex, endVertex);

return 0;

}

输入：请输入节点数目：8

请输入要添加的边的数量：10

请输入边的起始节点、目标节点和权重1 2 1

请输入边的起始节点、目标节点和权重1 3 1

请输入边的起始节点、目标节点和权重2 4 1

请输入边的起始节点、目标节点和权重3 6 1

请输入边的起始节点、目标节点和权重3 7 1

请输入边的起始节点、目标节点和权重4 8 1

请输入边的起始节点、目标节点和权重5 8 1

请输入边的起始节点、目标节点和权重6 8 1

请输入边的起始节点、目标节点和权重7 8 1

请输入边的起始节点、目标节点和权重2 5 1

输出：深度优先遍历结果：12485637广度优先遍历结果：12345678请输入最小路径的起点和终点：1 8

最小路径: 1 2 4 8

最小距离: 3

反思：

1.其实一开始dfs是不知道怎么做的，后来知道是邻接表的大小顺序使得遍历顺序不对，上网搜索后才使用尾插法确保插入节点的有序性。

2.至于bfs，在解决邻接表的有序性后就非常简单了。

3.一开始算法是不能遍历非连通图的，后来加了循环遍历未访问的节点就可以了