**北京邮电大学软件学院**

**2020-2021学年第1学期实验报告**

**课程名称： 算法与数据结构**

**实验名称：**  图及其应用

**实验完成人：**

**姓名：**\_\_\_杨沛霖\_\_\_**学号：**\_\_2019211976\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_\_\_贾红娓\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2020 年 12 月 1 日**

1. **实验目的**

树和图是两种应用极为广泛的数据结构，也是这门课程的重点。它们的特点在于非线性。 稀疏矩阵的十字链表存储结构也是图的一种存储结构。本章实验继续突出了数据结构加操作的程序设计观点，但根据这种结构的非线性特点，将操作进一步集中在遍历操作上，因为遍历操作是其它众多操作的基础。遍历逻辑的（或符号形式的）结构，访问动作可是任何操作。本次实验希望帮助学生熟悉各种存储结构的特征，以及如何应用图结构解决具体问题（即原理与应用的结合）。

1. **实验内容**

**（一）图遍历的演示**

**[问题描述]**

　　很多涉及图上操作的算法都是以图的遍历操作为基础的。试写一个程序，演示无向图的遍历操作。

**[基本要求]**

　　以邻接表为存储结构，实现连通无向图的深度优先和广度优先遍历。以用户指定的结点为起点，分别输出每种遍历下的结点访问序列和相应生成树的边集。

**[测试数据]**

　　由学生依据软件工程的测试技术自己确定。注意测试边界数据，如单个结点。

**[实现提示]**

设图的结点不超过30个，每个结点用一个编号表示（如果一个图有n个结点，则它们的编号分别为1,2,…,n）。通过输入图的全部边输入一个图，每条边为一个数对，可以对边的输入顺序做出某种限制。注意，生成树的边是有向边，端点顺序不能颠倒。

**（二）饿了么外卖最短路径推荐**

**[问题描述]**

外卖大家都不陌生，大家都想希望外卖配送的越快越好，外卖商家如何能够保证在规定时间配送，这是各大平台（饿了么等）关心的问题。每位同学需要实现以下外卖配送算法：外卖员小翔带着一批货物希望用自己仅剩的时间到达配送点。为了使问题简单，将交通路线假设为一个n\*m的迷宫，其中0代表路况良好可以通过，1表示交通拥堵或者路段无法通行。小翔初始位置为（0，0），配送地点为（0，m-1）（保证这两个位置都是0，并且一定有从起点到终点的路径存在），假设他横着走会消耗2点时间，向上会消耗3点时间，向下消耗1点时间。现在请你为配送系统实现算法，看小翔能否在仅剩的时间内到达配送点。

**[基本要求]**

使用DFS或者BFS解决问题。

**[测试数据]**

同学利用自己的软件测试知识自己生成测试数据，需要数据为两个：

1. 小翔的剩余时间；
2. 交通状况图，用一个二维数组表示；

**[实现提示]**

图的遍历算法。

1. **实验环境**

visual studio community 2019

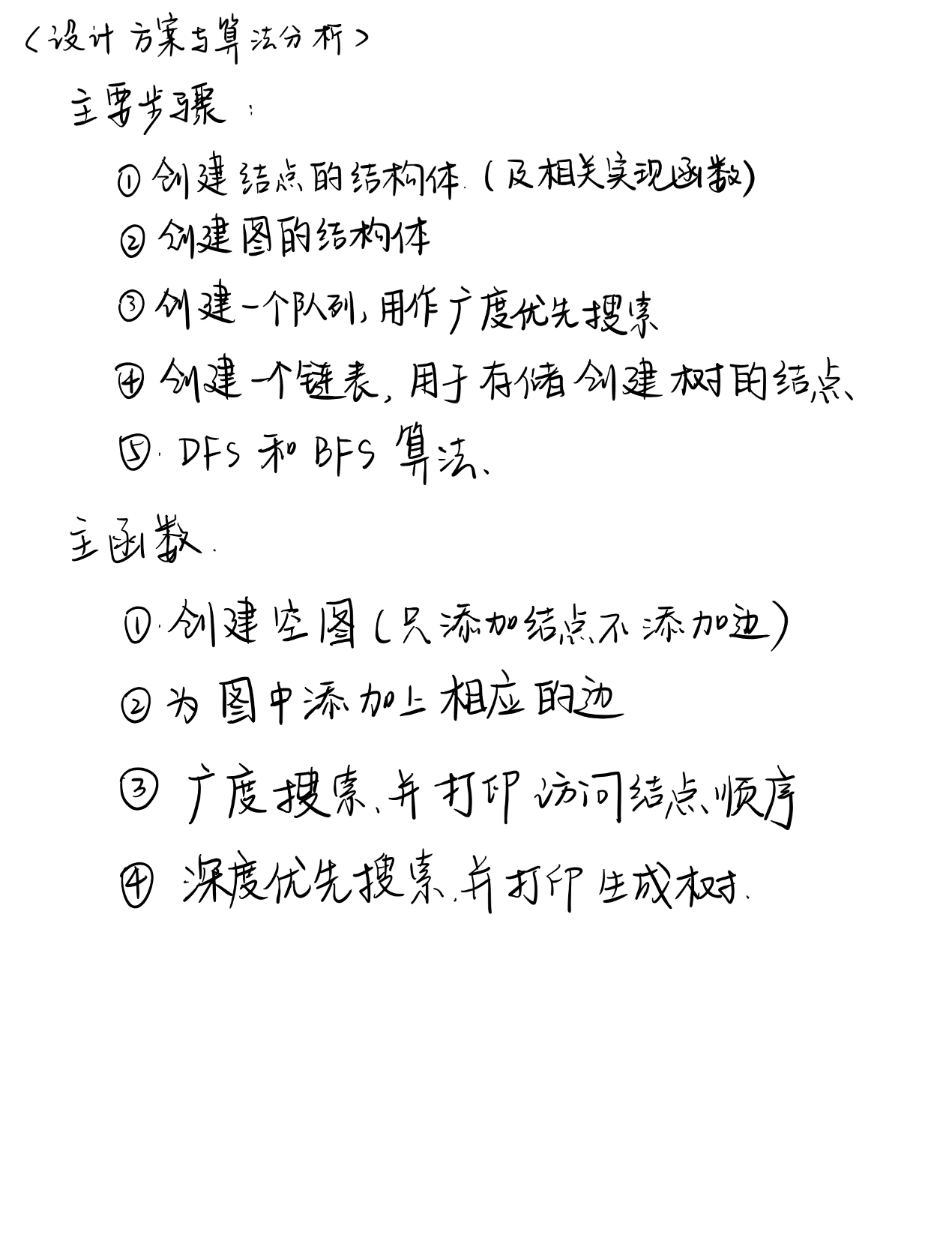
1. **实验过程和实验结果**

**实验一 图遍历的演示**

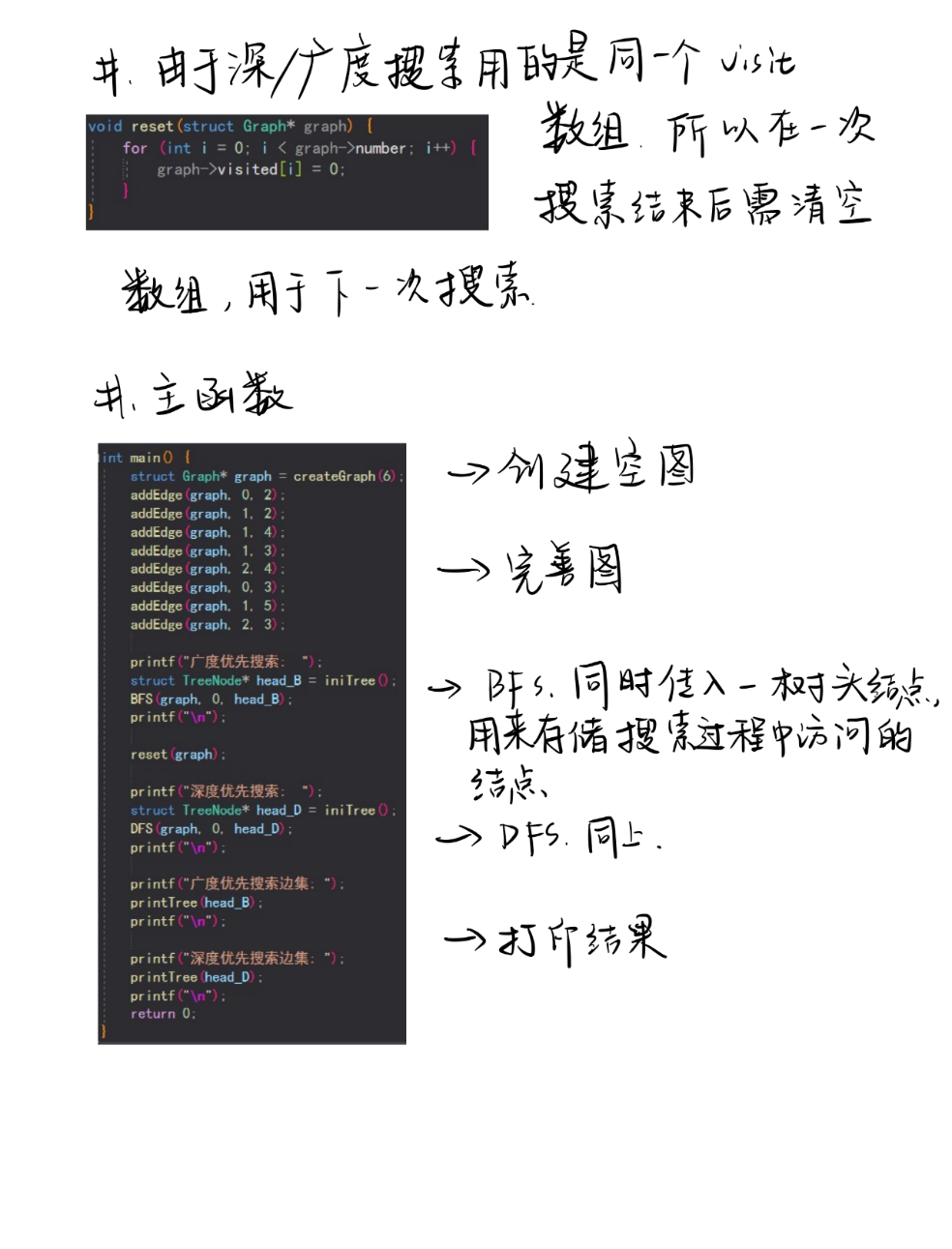
**问题分析:**

考察的是图的创建和遍历,以及在遍历的时候记录顺序,并且由这些结点的遍历顺序创建生成树

**设计方案与算法分析:**







**代码程序:**

#include <stdio.h>

#include <crtdbg.h>

#include <stdlib.h>

#define MAX 10000

#define \_CRT\_SECURE\_NO\_WARNINGS

#pragma warning(disable:4996)

/\*------------------------------------------图的结点---------------------------------------------------------\*/

struct Node {

int vertex;

struct Node\* next;

};

struct Node\* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

/\*------------------------------------图的结构体,里面有好多结点--------------------------------------------\*/

struct Graph {

struct Node\*\* List;

int number;

int\* visited;

};

struct Graph\* createGraph(int number) {

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

graph->number = number;

graph->List = (struct Node\*\*)malloc(number \* sizeof(struct Node\*));

graph->visited = (int\*)malloc(number \* sizeof(int));

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

graph->List[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

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

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node = createNode(dest);

node->next = graph->List[src];

graph->List[src] = node;

node = createNode(src);

node->next = graph->List[dest];

graph->List[dest] = node;

}

/\*---------------------------------------队列结点------------------------------------------------------\*/

struct QNode {

int data;

struct QNode\* next;

};

struct Queue {

struct QNode\* front, \* rear;

};

struct QNode\* newNode(int k){

struct QNode\* temp = (struct QNode\*)malloc(sizeof(struct QNode));

temp->data = k;

temp->next = NULL;

return temp;

}

struct Queue\* createQueue(){

struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

q->front = q->rear = NULL;

return q;

}

void enQueue(struct Queue\* q, int k){

struct QNode\* temp = newNode(k);

if (q->rear == NULL) {

q->front = q->rear = temp;

return;

}

q->rear->next = temp;

q->rear = temp;

}

int deQueue(struct Queue\* q){

if (q->front == NULL) {

return -1;

}

else {

struct QNode\* temp = q->front;

q->front = q->front->next;

if (q->front == NULL) {

q->rear = NULL;

}

int result = temp->data;

free(temp);

return result;

}

}

bool isEmpty(struct Queue\* q) {

return (q->rear == NULL);

}

void printQueue(struct Queue\* q) {

struct QNode\* node = q->front;

if (isEmpty(q)) {

printf("列表为空");

}

else {

while (node != NULL) {

printf("%d ", node->data);

node = node->next;

}

}

return;

}

/\*--------------------------------------存储树的链表------------------------------------------------------\*/

struct TreeNode {

int data;

struct TreeNode\* next;

};

struct TreeNode\* iniTree() {

struct TreeNode\* head = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

head->data = -1;

head->next = NULL;

// 返回头结点

return head;

}

struct TreeNode\* createTreeNode(int data) {

struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

node->data = data;

node->next = NULL;

return node;

}

void addTreeNode(struct TreeNode\* head,int data) {

struct TreeNode\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

struct TreeNode\* newNode = createTreeNode(data);

newNode->next = NULL;

temp->next = newNode;

return;

}

void printTree(struct TreeNode\* head) {

struct TreeNode\* temp = head->next;

while (temp != NULL) {

printf("%d", temp->data);

if (temp->next != NULL) {

printf("->");

}

temp = temp->next;

}

}

/\*--------------------------------------DFS深度搜索-----------------------------------------------------\*/

void DFS(struct Graph\* graph, int vertex, struct TreeNode\* head) {

struct Node\* List = graph->List[vertex];

struct Node\* temp = List;

graph->visited[vertex] = 1;

addTreeNode(head,vertex);

printf("%d ", vertex);

while (temp != NULL) {

int nextVertex = temp->vertex;

if (graph->visited[nextVertex] == 0) {

DFS(graph, nextVertex, head);

}

temp = temp->next;

}

}

/\*-----------------------------广度优先搜索-----------------------\*/

void BFS(struct Graph\* graph, int vertex,struct TreeNode\* head) {

struct Queue\* q = createQueue();

graph->visited[vertex] = 1;

enQueue(q, vertex);

while (!isEmpty(q)) {

int currentVertex = deQueue(q);

addTreeNode(head, currentVertex);

printf("%d ", currentVertex);

struct Node\* temp = graph->List[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enQueue(q, adjVertex);

}

temp = temp->next;

}

}

}

/\*------------------------重置visited数组----------------------------\*/

void reset(struct Graph\* graph) {

for (int i = 0; i < graph->number; i++) {

graph->visited[i] = 0;

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 0, 3);

addEdge(graph, 1, 5);

addEdge(graph, 2, 3);

printf("广度优先搜索: ");

struct TreeNode\* head\_B = iniTree();

BFS(graph, 0, head\_B);

printf("\n");

reset(graph);

printf("深度优先搜索: ");

struct TreeNode\* head\_D = iniTree();

DFS(graph, 0, head\_D);

printf("\n");

printf("广度优先搜索边集: ");

printTree(head\_B);

printf("\n");

printf("深度优先搜索边集: ");

printTree(head\_D);

printf("\n");

return 0;

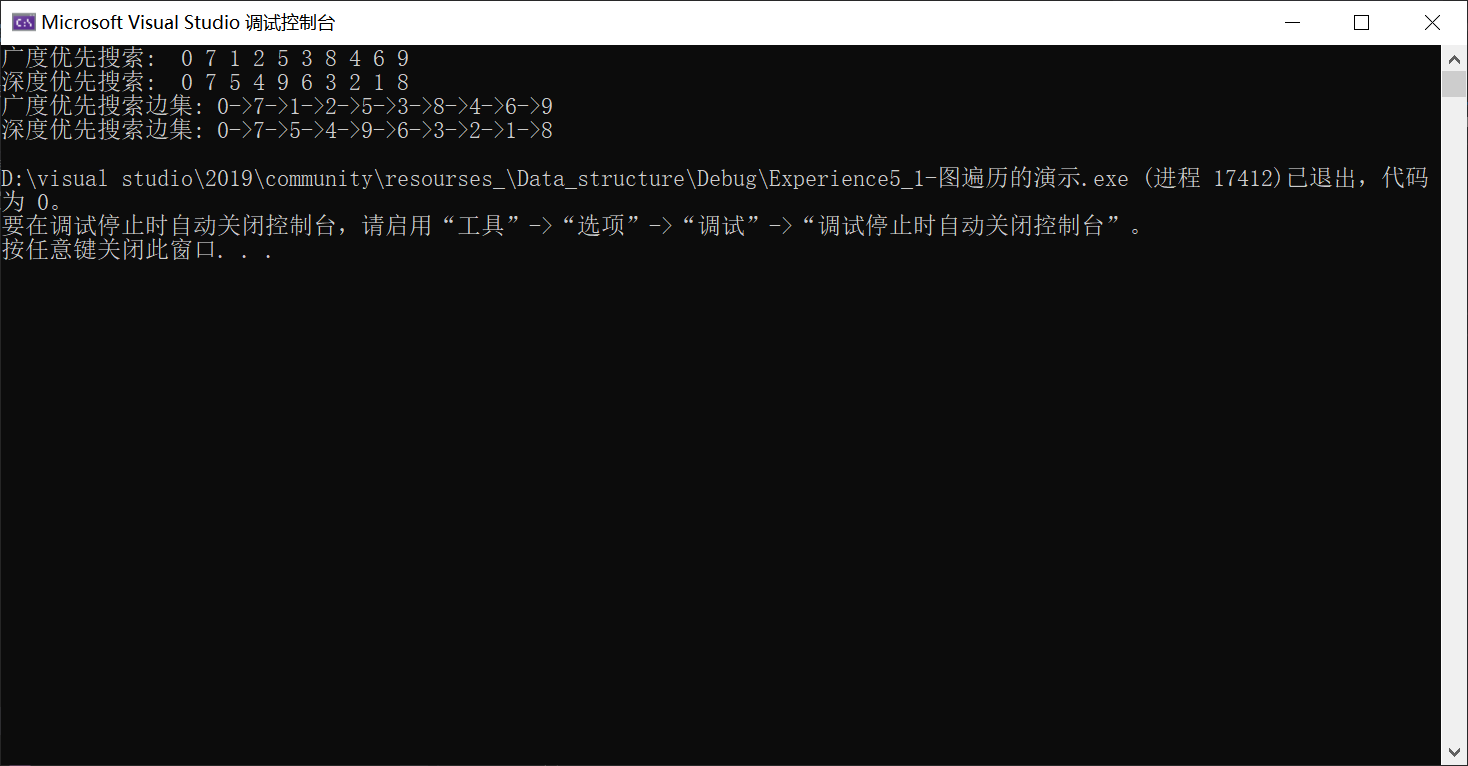
}

**调试和验证过程:**

1. DFS用的是递归遍历,也就是访问一个结点,之后要访问这个结点所连接的第一个结点,之后再访问这个所连接的第一个结点所连接的第一个节点…直到一个结点没有连接的下一个结点或者所连接的下一个结点已经在这次遍历中被访问到,之后回退,寻找某一个结点所连接的第二个结点(该节点没有被访问过),并且继续这个过程.
2. BFS不需要递归,但是需要一个队列,把第一个结点放入队列,之后开始一个while循环,如果队列不为空就从队列中拿出一个结点(访问该结点),之后把这个结点所连接到的所有节点再次放入队列,重复循环.操作更复杂但是不需要递归.
3. 在遍历的时候顺带传进去一个树的根节点,存储遍历的结点顺序.

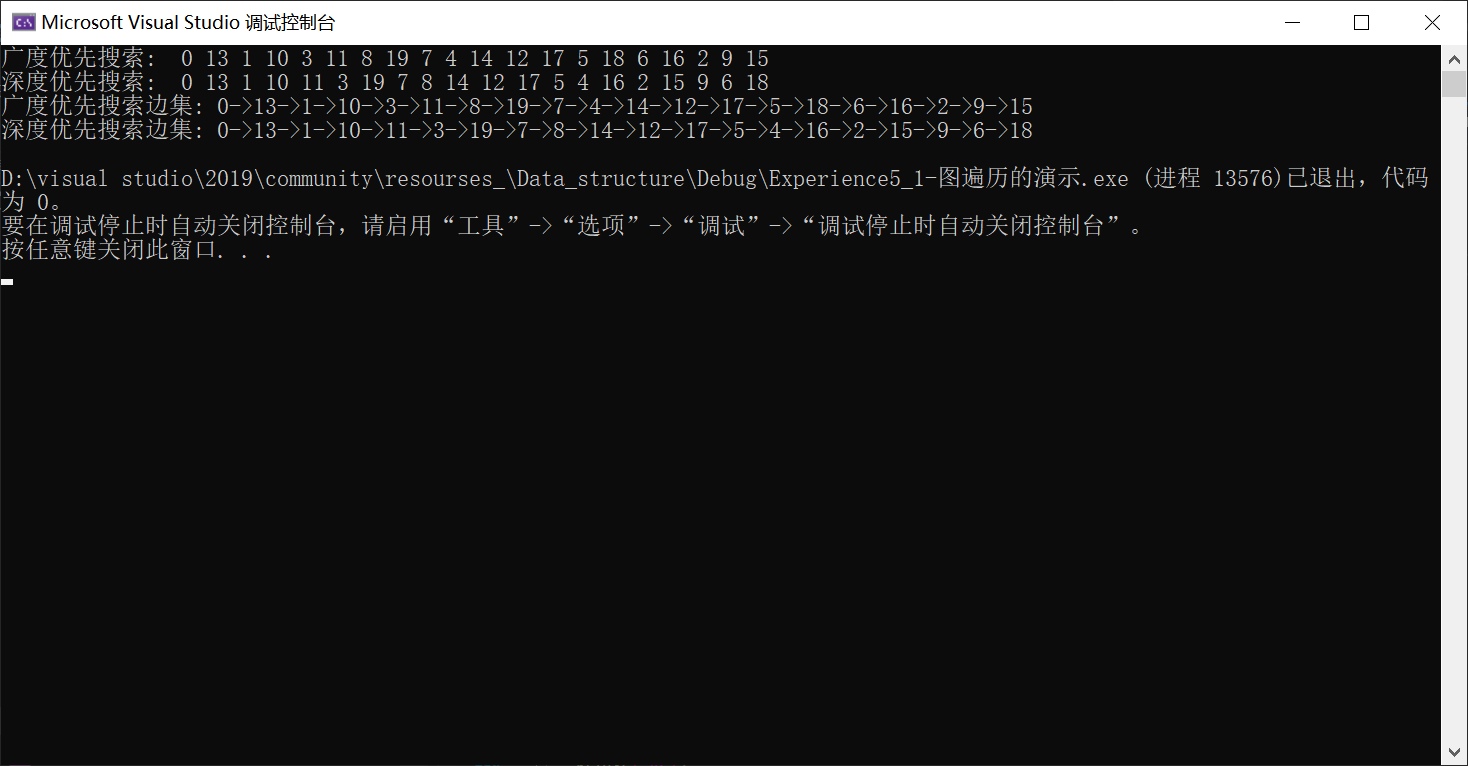
**运行结果分析:**





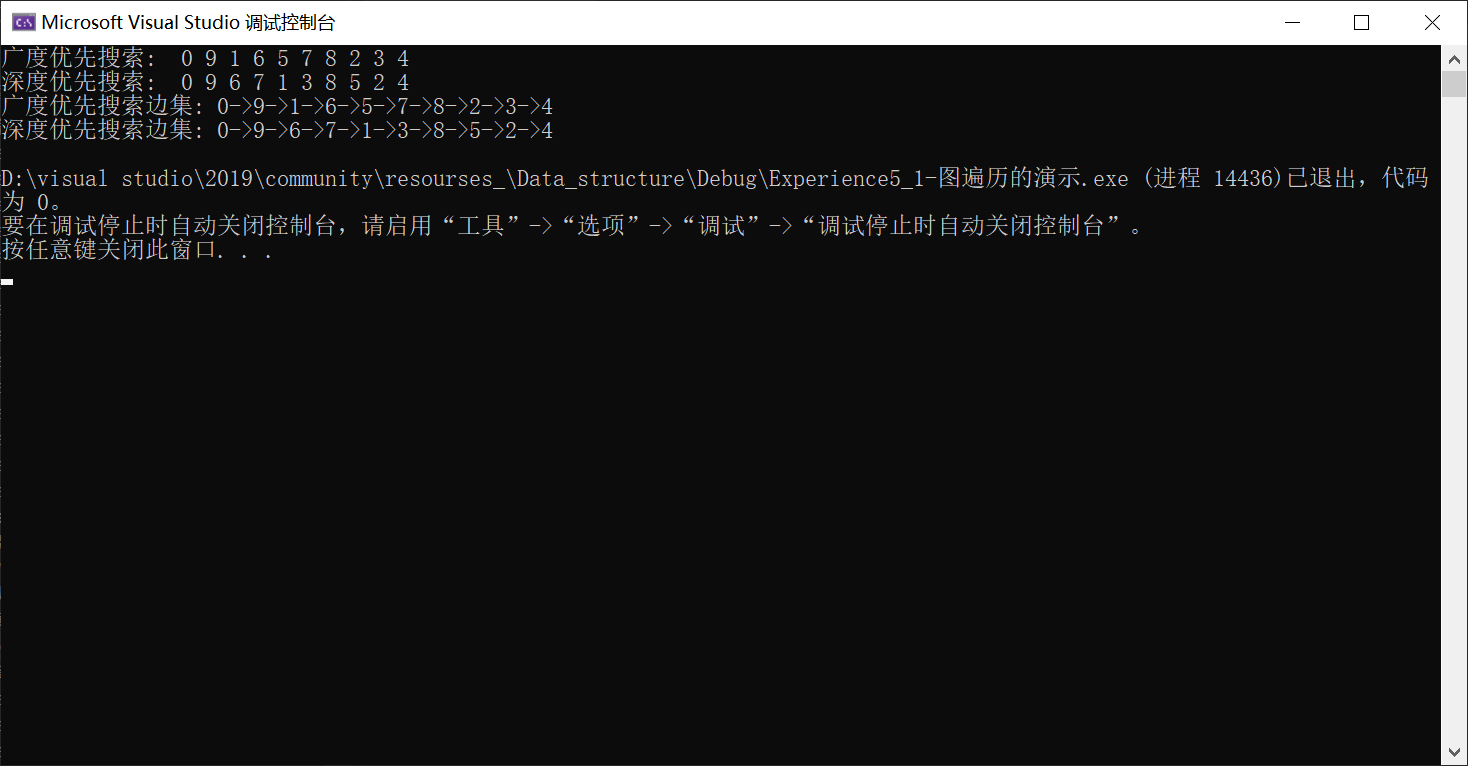
---------------------------分割线----------------------------





---------------------------分割线----------------------------





---------------------------分割线----------------------------





---------------------------分割线-----------------------------



---------------------------分割线-----------------------------

**实验二 饿了么外卖最短路径推荐**

**问题分析:**

是从确定一点到另外一确定点之间,由于不同的方向选择会造成不同的时间消耗,从而找到最短时间消耗的路径.(本题不要求打印出路径,实际上,只需要记录最短时间用来比较就可以)

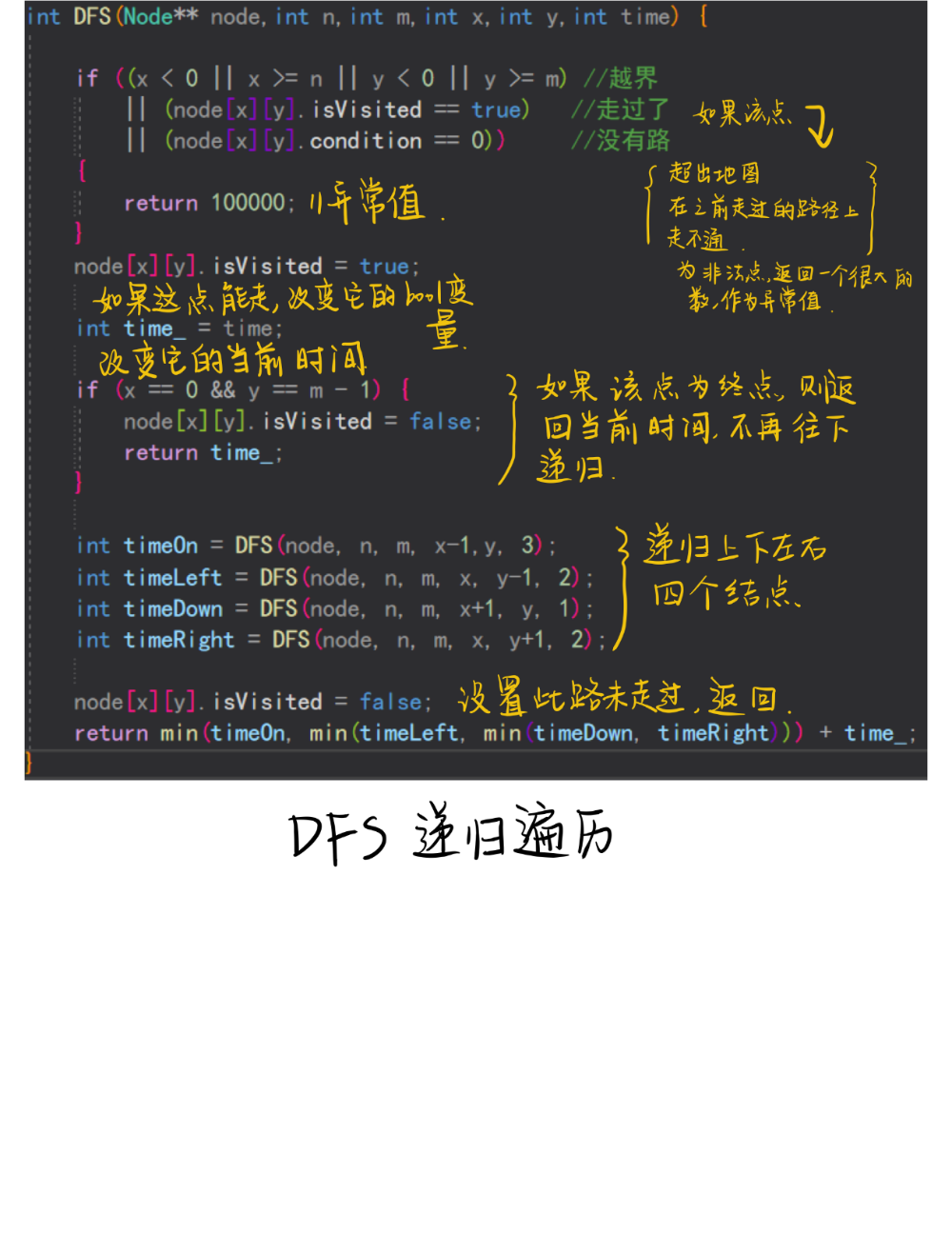
**设计方案与算法分析:**

1. 首先输入n和m,之后创建相应的二维矩阵,输入时间
2. 创建结构体,结构体存储的是
   1. condition->int类型: 这个点所存的数字(0或1)
   2. isVisited->bool类型: 该点是否被访问过
   3. time ->int类型: 从该点开始走到终点的最短时间
   4. x->int类型: 该点所在二维数组中的x坐标(从0到n-1)
   5. y->int类型: 该点所在二维数组中的y坐标(从0到m-1)
3. 编写DFS算法

总体思想:

递归遍历,DFS接收的参数主要是一个结点和一个时间(int),返回值是从这个结点开始,走到终点的最短时间.这个结点会递归调用它的上下左右四个节点的DFS,共返回从这个节点的上下左右结点作为起点走到终点的最短时间,之后选取这四个时间中最小的时间,加上自己被传入的时间,作为自己的返回值.

具体分析:



**程序:**

#include<iostream>

#include <algorithm>

#define MAX 100

using namespace std;

class Node {

public:

int time;

bool isVisited;

int condition;

int x;

int y;

};

int DFS(Node\*\* node,int n,int m,int x,int y,int time) {

if ((x < 0 || x >= n || y < 0 || y >= m) //越界

|| (node[x][y].isVisited == true) //走过了

|| (node[x][y].condition == 0)) //没有路

{

return 100000;

}

node[x][y].isVisited = true;

int time\_ = time;

if (x == 0 && y == m - 1) {

node[x][y].isVisited = false;

return time\_;

}

int timeOn = DFS(node, n, m, x-1,y, 3);

int timeLeft = DFS(node, n, m, x, y-1, 2);

int timeDown = DFS(node, n, m, x+1, y, 1);

int timeRight = DFS(node, n, m, x, y+1, 2);

node[x][y].isVisited = false;

return min(timeOn, min(timeLeft, min(timeDown, timeRight))) + time\_;

}

int main() {

Node\*\* node = nullptr;

int n, m;

cout << "请分别输入n和m的值: " << endl;

cin >> n >> m;

int time = 0;

cout << "请输入剩余的时间: " << endl;

cin >> time;

node = (class Node\*\*)malloc(n\* sizeof(class Node\*));

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

node[i] = (class Node\*)malloc(m \* sizeof(class Node));

}

cout << "请分别输入 " << n << "行" << m << "列的二维数组,用0代表此路不通,用1代表此路畅通" << endl;

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

for (int j = 0; j < m; j++) {

cin >> node[i][j].condition;

node[i][j].x = i;

node[i][j].y = j;

node[i][j].isVisited = false;

}

}

int result = DFS(node,n,m,0,0,0);

cout << "所需配送最短时间为: " << result << endl;

if (result < time) {

cout << "时间充足" << endl;

}

else if (result == time) {

cout << "时间刚刚好" << endl;

}

else {

cout << "时间不够" << endl;

}

return 0;

}

**调试和验证过程:**

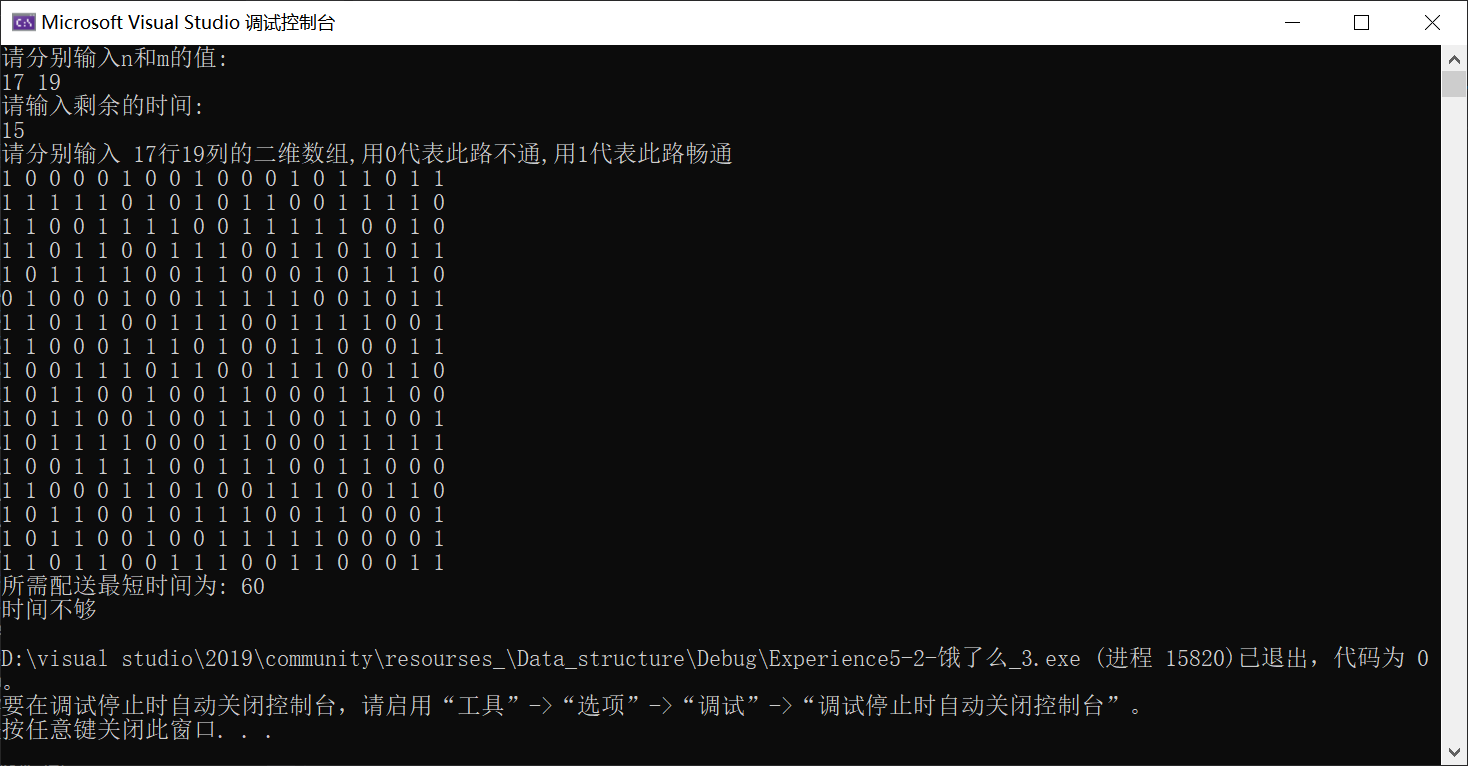
1. 这道题其实可以不用邻接表或者邻接矩阵,一个是因为图的样式已经给你了,二是因为建立图结构的话很麻烦而且效率得不到提高.我用了一个类(结构体),可以多存储几个信息,本质上还是在二维数组中直接遍历了.
2. 要注意x和y对应的n和m的次序,不要搞乱
3. 如果访问到终点的话就直接返回time,因为从终点到终点的最短时间为0,不管上下左右走的时间都会比0长.
4. 要注意在什么位置改变isVisited的值.
5. 在return之前要把isVisited的值改变一下,为的是下面的路径可以访问到这一点.(为的是走过所有路径)

**运行结果分析:**









1. **实验心得**

图的代码主要有这么几种

1. 生成图(用边)
2. 深度搜索
3. 广度搜索
4. Prim算法
5. Kruskal算法

其中深度搜索和广度搜索是最重要的.

图比树更加复杂,希望老师可以留更多的作业,锻炼我们的代码能力,我觉得第二题选做的话作业量有点太少了.希望自己能够能进一步,期末加油.