### 第3题：特殊路径统计

#### 数据结构：

采用了队列（queue）来实现BFS

采用邻接表来实现图

非表头结点机构体（ArcNode）

typedef struct ArcNode {

int adjvex;

struct ArcNode\* nextarc;

}ArcNode;

表头结点 结构体(VexNode)

typedef struct VexNode {

int maxdata; //所在路径上的最大值，判断是否为最大路径

ArcNode\* firstarc;

}VexNode;

邻接表结构体（Graph）

typedef struct Graph { //邻接表结构类型

VexNode\* VNode; //定义邻接表

int vexnum; //顶点个数

}Graph;

#### 算法设计思想：

首先调用CreatGraph()函数获取创建邻接表。然后调用CountRoad()函数获取特殊路径数，该函数中又对一个个点调用BFS（）函数中找到并输出特殊路径。

在BFS函数中，先保存当前路径的最大值，然后在拓展时判断当前结点是否比路径起点小，若小，则不让该点加入BFS辅助拓展队列中；若大，则判断是否比当前路径最大值小。

若比当前路径最大值小，则加入拓展队列中，若大，则为一条新的特殊路径，输出该路径并更新最大路径最大值。

#### 源程序：

#include <iostream>

#include <queue>

#include <stdlib.h>

#include <Windows.h>

using namespace std;

#define red SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED) //红色

#define yellow SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN) //黄色

#define green SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_GREEN) //绿色

#define white SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE) //白色

#define ching SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_GREEN |FOREGROUND\_BLUE) //青色

void INFO(string info) {

cout << "["; green; cout << "INFO"; white; cout << "]:" << info ;

}

void ERROROUT(string error) {

cout << "["; red; cout << "ERROR"; white; cout << "]:" << error << endl;

}

//以邻接表为数据结构

//非表头结点

typedef struct ArcNode {

int adjvex;

struct ArcNode\* nextarc;

}ArcNode;

//表头结点

typedef struct VexNode {

//int data;

int maxdata; //所在路径上的最大值，判断是否为最大路径

ArcNode\* firstarc;

}VexNode;

typedef struct Graph { //邻接表结构类型

VexNode\* VNode; //定义邻接表

int vexnum; //顶点个数

}Graph;

int \*visit; //标记点是否被访问过，动态申请

int count = 0; //特殊路径总数

int num; //顶点个数

int lastSecondeRoad = 0; //第二大的点的特殊路径数

int isConnectedlast = 0; //第一大的点与第二大的点是否联通

void CreatGraph(Graph &G){ //动态申请空间建图

while (1){

INFO("请输入顶点个数\n");

cin >> num;

//顶点个数小于2错误

if (num <= 1)

ERROROUT("顶点个数过少，不能构成路径,请重新输入顶点个数\n");

else

break;

}

if (num == 2){ //顶点个数为2直接得路径数不必建图

int fake,fakes;

INFO("请输入相关信息：\n");

cin >> fake; cin >> fakes;

return ;

}

else {

G.vexnum = num;

G.VNode = (VexNode\*)malloc ((num+1)\* sizeof(VexNode));

}

//为访问数组动态分配空间

visit = (int\*)malloc((num+1) \* sizeof(int));

//邻接表初始化，所有单向链表均为空表

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

//G.VNode[i].data = i+1;

G.VNode[i].firstarc = NULL;

}

int a;

ArcNode \*p,\*q;

INFO("请输入相关信息：\n");

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

cin >> a;

//判断第一大的点和第二点是否联通

if ((i == G.vexnum-1 && a == G.vexnum)||(a == G.vexnum-1 && i == G.vexnum))

isConnectedlast = 1;

if (a > 0) {

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

p->nextarc = NULL;

p->adjvex = i;

q = G.VNode[a].firstarc;

if (q == NULL) {

G.VNode[a].firstarc = p;

}

else {

while (q->nextarc != NULL) {

q = q->nextarc;

}

q->nextarc = p;

}

//无向图创建对称边

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

p->nextarc = NULL;

p->adjvex = a;

q = G.VNode[i].firstarc;

if (q == NULL) {

G.VNode[i].firstarc = p;

}

else {

while (q->nextarc != NULL) {

q = q->nextarc;

}

q->nextarc = p;

}

}

}

}

void BFS(Graph G,int i){ //BFS获取路径

queue<int> q;

visit[i] = 1;

q.push(i);

int shouldNewline = 0;

int maxInRoad;

while (!q.empty()){

maxInRoad = q.front();

q.pop();

ArcNode \*p = G.VNode[maxInRoad].firstarc;

while (p){

if (!visit[p->adjvex] && p->adjvex >= i) {

if (p->adjvex >= G.VNode[maxInRoad].maxdata){

//若当前结点大于当前路径的最大值，更新

count++;

cout << i << "->" << p->adjvex << " ";

shouldNewline = 1;

}

else{

G.VNode[p->adjvex].maxdata = G.VNode[maxInRoad].maxdata;

}

visit[p->adjvex] = 1;

q.push(p->adjvex);

}

p = p->nextarc;

}

}

if (shouldNewline){

cout << endl;

}

}

int CountRoad (Graph &G){ //计算特殊路径

INFO("特殊路径为：\n") ;

if (num == 2){

cout <<"1->2"<<endl;

count = 2; //顶点个数为2直接得出结果

}

else{

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

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

visit[i] = 0; //辅助数组Visit初始化

G.VNode[i].maxdata = i; //当前结点所在路径上的最大值初始化

}

BFS(G, i);

}

if (isConnectedlast){

count++;

cout <<G.vexnum-1<<"->"<<G.vexnum<<endl;

}

}

INFO("特殊路径数为：");cout<<count<<endl;

}

int main(){

Graph G;

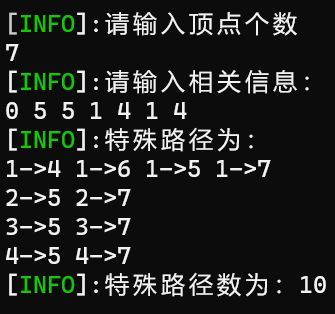
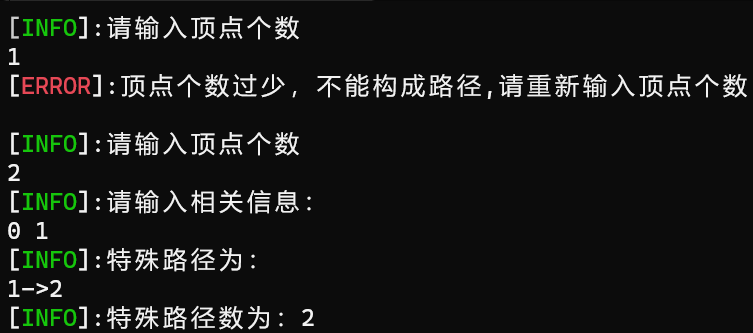
CreatGraph(G);

CountRoad(G);

return 0;

}

#### 测试数据和结果



#### 时间复杂度：

BFS的时间复杂度为O（n^2）

如果顶点个数为2，则为O（1）

#### 改进：

首先，若顶点个数小于2，则不可能有特殊路径。

其次，若顶点个数为2，特殊路径肯定只有一个，即1->2。那就不必创建图和进行BFS遍历了，若出现这种情况的时间复杂度为O（1）

然后，肯定不会出现路径起点为**最大的数（biggest，以下简称B）**，故不对最大的数进行BFS遍历，减少了0-3次的拓展时间。

并且，路径起点为**第二大的数（second biggest，以下简称SB）**.

路径数要么为0，要么为 1: SB-> B. 那么也不需要对第二大的店进行BFS拓展，只需要判断它与最大的数是否联通即可，这个可以在输入父结点信息时就可以判断。减少了最少1次，最多3次的BFS拓展。

#### 反思：

其实，若起点为**第三大的数（简称TB）**，还可以继续改进：根据之前获取的B与SB是否相连，若B与SB相连，则TB的可成路径邻接点肯定只能要么B，要么SB，是B特殊路径有1条，是SB特殊路径有两条。若B与SB，判断TB的邻接点里有B和SB这两个的几个，有几个就有几条特殊路径。

如果是对TB常规BFS扩展，当TB的邻接点里有SB时，还需要SB进行扩展，增加了额外的不必要的拓展。

此外，该程序的CreateGraph函数和 CountRoad函数，共同使用的变量较多，在程序里我将这些设为全局变量来减少函数传参个数，恰当的办法是将它们（共同使用的变量和函数）封装为一个类。

这两个函数的缺点还有一个就是耦合度较高，以后编程时我要注意到这个问题。

#### 该题代码行：

166行