**Project 6实验报告**

**22336313 郑鸿鑫**

1. **程序功能简要说明。**

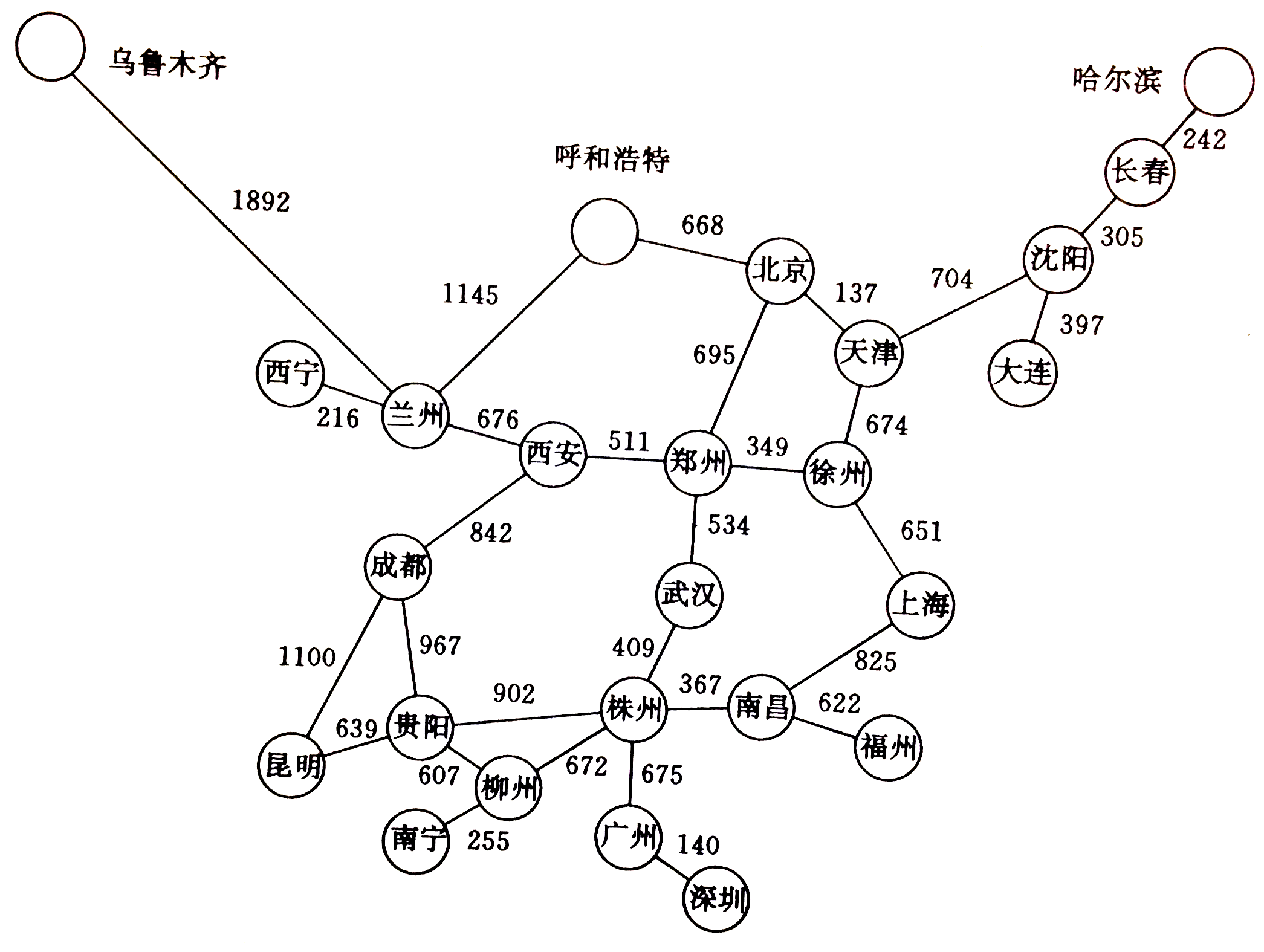
本程序实现对无向图的遍历，以两种不同的存储方式来存储无向图，主要实现以下功能：  
1.以邻接多重表为存储结构，可以实现无向图的深度优先和广度优先遍历，可以输入指定的节点作为起点，分别输出节点的访问序列和相应生成树的边序列。

2.用非递归算法实现深度优先搜索。

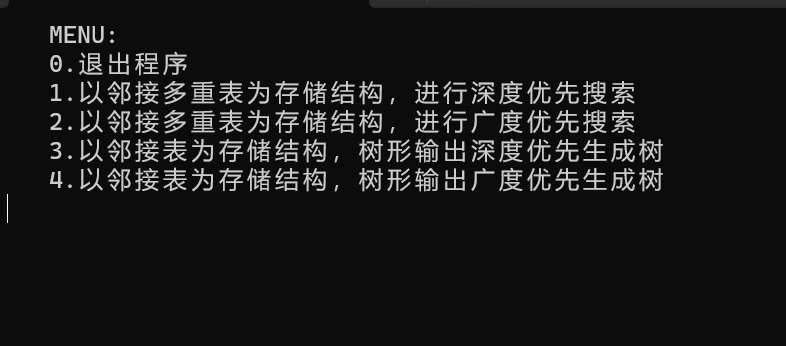
3.以邻接表为存储结构，建立深度优先生成树和广度优先生成树，并以树形输出生成树

1. **程序运行截图，包括计算功能演示、部分实际运行结果展示、命令行或交互式界面效果等。**

测试数据：（存储在txt文件中，以文件输入作为输入）：



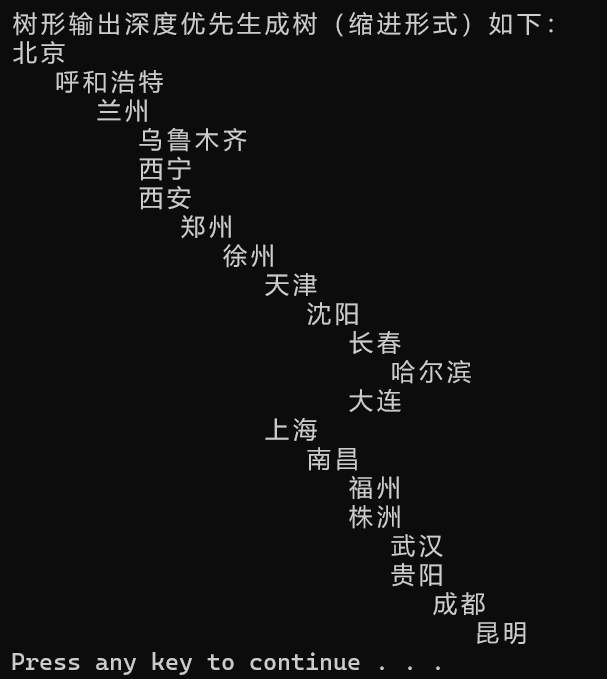
**程序功能选单界面：**

****

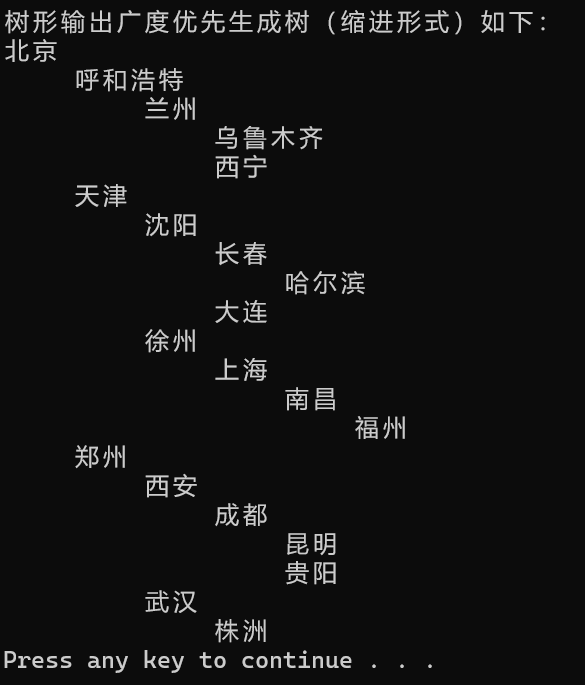
**功能1以邻接多重表为存储结构，进行深度优先搜索：  
**

**功能2以邻接多重表为存储结构，进行广度优先搜索：**

****

**功能3以邻接表为存储结构，树形输出深度优先生成树：（以北京为起点）  
**

**功能4以邻接表为存储结构，树形输出广度优先生成树：（以北京为起点）**

****

1. **部分关键代码及其说明。**

**自行定义并实现栈：**

#ifndef **PROJECT6\_MYSTACK\_H**#define **PROJECT6\_MYSTACK\_H**#pragma once  
class MyStack  
{  
public:  
 int \*m\_pBuffer; // 栈空间的指针  
 int m\_iSize; // 栈容量  
 int m\_iTop; // 栈顶  
 MyStack(int size);  
 ~MyStack();  
 bool stackEmpty(); //判断栈是否为空  
 bool stackFull(); //判断栈是否满  
 bool push(int elem); // 将元素入栈  
 bool pop(); // 出栈  
};  
MyStack::MyStack(int size)  
{  
 m\_iSize = size;  
 m\_pBuffer = new int[size];  
 m\_iTop = 0;  
}  
MyStack::~MyStack()  
{  
 delete[] m\_pBuffer;  
}  
bool MyStack::stackEmpty()  
{  
 if (m\_iTop == 0)  
 return true;  
 else  
 return false;  
}  
bool MyStack::stackFull()  
{  
 if (m\_iTop >= m\_iSize)  
 return true;  
 else  
 return false;  
}  
bool MyStack::push(int elem)  
{  
 if (stackFull())  
 {  
 return false;  
 }  
 m\_pBuffer[m\_iTop] = elem;  
 m\_iTop++;  
 return true;  
}  
bool MyStack::pop()  
{  
 if (stackEmpty())  
 return false;  
 m\_iTop--;  
 return true;  
}  
#endif //PROJECT6\_MYSTACK\_H

**邻接多重表定义：**

class Edge{  
public:  
 int vexi;  
 int vexj;  
 Edge\* linki;  
 Edge\* linkj;  
 int weight;  
 Edge(int i,int j,int wei):vexi(i),vexj(j),weight(wei),linki(nullptr),linkj(nullptr){}  
};  
class Vex{  
public:  
 string place;  
 Edge\* Fout;  
 Vex(string p):place(p),Fout(nullptr){}  
};  
struct AMLGraph{  
public:  
 vector<Vex> adj;  
 int vexnum;//顶点数  
 int edgenum;//边数  
};

**非递归深搜和广搜：**

void DFS\_AML(AMLGraph &g, int s) {  
 int n = g.vexnum;  
 bool visited[n];  
 for(int i = 0;i < n;i++){  
 visited[i] = false;  
 }  
 MyStack sta(30); // 存储顶点索引的栈  
 sta.push(s); // 将起始顶点入栈  
 visited[s] = true;  
 cout << setw(10) <<g.adj[s].place << " "; int i =1;  
 while(!sta.stackEmpty()){  
 int idx = sta.m\_pBuffer[sta.m\_iTop-1];  
 Edge\* e = g.adj[idx].Fout;  
 bool flag = false;  
 while(e){  
 int nextIdx = (e->vexi == idx) ? e->vexj : e->vexi; // 获取邻接顶点的索引  
 if(!visited[nextIdx]){  
 cout << "第" << i++ << "条边为： " << g.adj[e -> vexi].place << " - " << g.adj[e -> vexj].place<< " 它的权重为： "<< e->weight << endl;  
 cout << setw(10) << g.adj[nextIdx].place << " "; // 访问顶点  
 sta.push(nextIdx); // 将未访问的邻接顶点入栈  
 visited[nextIdx] = true; // 标记为已访问  
 flag = true;  
 break;  
 }  
 e = (e->vexi == idx) ? e->linki : e->linkj; // 获取下一条边  
 }  
 if(!flag){  
 sta.pop(); // 如果当前顶点的所有邻接顶点都已访问过，出栈  
 }  
 }  
 cout << endl;  
}  
void BFS\_AML(AMLGraph &g, int s) {  
 int n = g.vexnum;  
 bool visited[n];  
 for(int i = 0;i < n;i++){  
 visited[i] = false;  
 }  
 queue<int>q;  
 q.push(s);  
 visited[s] = true;  
 cout << setw(10) << g.adj[s].place << ' ';  
 int i = 1;  
 while(!q.empty()){  
 int idx = q.front();  
 q.pop();  
 Edge\* e = g.adj[idx].Fout;  
 while(e){  
 int nextIdx = (e->vexi == idx) ? e->vexj : e->vexi; // 获取邻接顶点的索引  
 if(!visited[nextIdx]){  
 cout << "第" << i++ << "条边为： " << g.adj[e -> vexi].place << " - " << g.adj[e -> vexj].place<< " 它的权重为： "<< e->weight << endl;  
 cout << setw(10) << g.adj[nextIdx].place<< " ";  
 visited[nextIdx] = true;  
 q.push(nextIdx);  
 }  
 e = (e->vexi == idx) ? e->linki : e->linkj; // 获取下一条边  
 }  
 }  
 cout << endl;  
}

**邻接表定义：**

struct vex{  
 string data;  
 vector<int>adj;  
 vector<vex\*> children;  
};  
struct ADgraph{  
 int vnum;  
 vector<vex>list;  
 void addEdge(int v,int w){  
 list[v].adj.push\_back(w);  
 list[w].adj.push\_back(v);  
 };  
};

**深度优先生成树和广度优先生成树树形输出：**

void DFS\_AD(ADgraph& g,vector<bool>& visited,int s,vector<int>& parent,int depth){  
 for(int i = 0;i < depth;i++){  
 cout << " ";  
 }  
 cout << g.list[s].data << endl;  
 visited[s] = true;  
 for(int i:g.list[s].adj){  
 if(!visited[i]) {  
 parent[i] = s;  
 DFS\_AD(g, visited, i, parent, depth + 1);  
 }  
 }  
}  
void BFS\_AD(ADgraph&g,int s){  
 vector<bool>visited(g.vnum,false);  
 queue<int> q;  
 q.push(s);  
 visited[s] = true;  
 while(!q.empty()) {  
 int x = q.front();  
 q.pop();  
 for (int i: g.list[x].adj) {  
 if (!visited[i]) {  
 q.push(i);  
 g.list[x].children.push\_back(&g.list[i]);  
 visited[i] = true;  
 }  
 }  
 }  
}  
void BFSTree(ADgraph&g,vex\* s,int depth){  
 for(int i = 0;i < depth;i++){  
 cout << " ";  
 }  
 cout << s->data << endl;  
 for(auto it:s->children) {  
 BFSTree(g,it,depth+1);  
 }  
}

**4、程序运行方式简要说明。**

程序通过输入数字选择不同功能，在功能1 2中需要输入起始节点，在功能3 4 中以北京为起点为例直接打印树形输出（以缩进形式）。

最后键入0以退出程序。