22336216-陶宇卓-Project6-实验报告

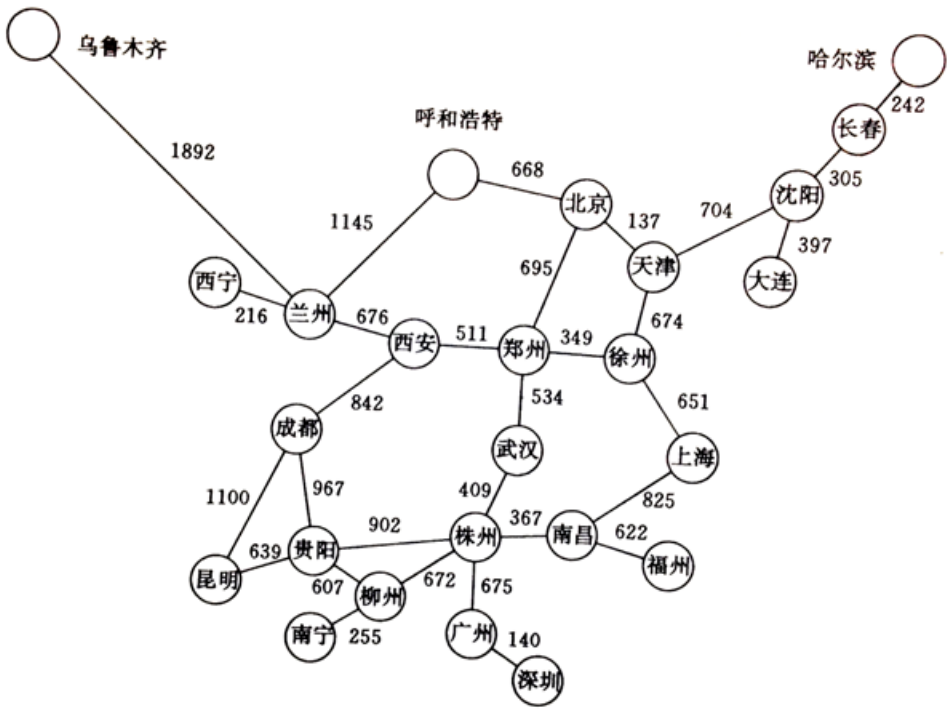
程序功能简要说明

(1) 以邻接多重表为存储结构，实现联通无向图的深度优先和广度优先遍历。以指定的结点为起点，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。

(2) 借助于栈类型（自行定义和实现），用非递归算法实现深度优先遍历。

(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。

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



以此为测试样例：

```
选择功能:
1. 深度优先搜索 (DFS)
2. 非递归深度优先搜索 DFS
3. 广度优先搜索 (BFS)
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 1
输入起点:6
DFS from vertex 北京 :
北京 天津 沈阳 长春 哈尔滨 大连 徐州 上海 南昌 福州 株洲 广州 深圳 贵阳 柳州 南宁 昆明 成都 西安 郑州 武汉 兰州 呼和浩特 西宁 乌鲁木齐
( 6 , 8 ) ( 8 , 10 ) ( 10 , 12 ) ( 12 , 13 ) ( 10 , 11 ) ( 8 , 9 ) ( 9 , 16 ) ( 16 , 20 ) ( 20 , 21 ) ( 20 , 19 ) ( 19 , 24 ) ( 24 , 25 ) ( 19 , 18 )
( 18 , 23 ) ( 23 , 22 ) ( 18 , 17 ) ( 17 , 14 ) ( 14 , 5 ) ( 5 , 7 ) ( 7 , 15 ) ( 5 , 3 ) ( 3 , 4 ) ( 3 , 2 ) ( 3 , 1 )
请按任意键继续. . .
```

```
选择功能:
1. 深度优先搜索 (DFS)
2. 非递归深度优先搜索 DFS
3. 广度优先搜索 (BFS)
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 2
输入起点:6
Non-recursive DFS from vertex 北京:
北京 天津 沈阳 长春 哈尔滨 大连 徐州 上海 南昌 福州 株洲 广州 深圳 贵阳 柳州 南宁 昆明 成都 西安 郑州 武汉 兰州 呼和浩特 西宁 乌鲁木齐
请按任意键继续. . .
```

```
选择功能:
1. 深度优先搜索 (DFS)
2. 非递归深度优先搜索 DFS
3. 广度优先搜索 (BFS)
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 3
输入起点:6
BFS starting from vertex 北京:
北京 天津 郑州 呼和浩特 沈阳 徐州 武汉 西安 兰州 长春 大连 上海 株洲 成都 西宁 乌鲁木齐 哈尔滨 南昌 广州 贵阳 昆明 福州 深圳 柳州 南宁

( 6 , 8 ) ( 6 , 7 ) ( 6 , 4 ) ( 8 , 10 ) ( 8 , 9 ) ( 7 , 15 ) ( 7 , 9 ) ( 7 , 5 ) ( 4 , 3 ) ( 10 , 12 ) ( 10 , 11 ) ( 9 , 16 ) ( 15 , 19 ) ( 5 , 14 )
( 5 , 3 ) ( 3 , 2 ) ( 3 , 1 ) ( 12 , 13 ) ( 16 , 20 ) ( 19 , 24 ) ( 19 , 20 ) ( 19 , 18 ) ( 14 , 18 ) ( 14 , 17 ) ( 20 , 21 ) ( 24 , 25 ) ( 18 , 23 )
( 18 , 17 ) ( 23 , 22 )
请按任意键继续. . .
```

```
选择功能:
1. 深度优先搜索 (DFS)
2. 非递归深度优先搜索 DFS
3. 广度优先搜索 (BFS)
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 4
输入起点:6
DFS Tree starting from vertex 北京:
|---北京
|------天津
|---------沈阳
|------------长春
|---------------哈尔滨
|------------大连
|---------徐州
|------------上海
|---------------南昌
|------------------福州
|------------------株洲
|--------------------广州
|----------------------深圳
|--------------------贵阳
|----------------------柳州
|--------------------------南宁
|----------------------昆明
|--------------------------成都
|----------------------------西安
|--------------------------------郑州
|----------------------------------武汉
|--------------------------------兰州
|----------------------------------呼和浩特
|----------------------------------西宁
|----------------------------------乌鲁木齐
```

```
选择功能:
1. 深度优先搜索（DFS）
2. 非递归深度优先搜索 DFS
3. 广度优先搜索（BFS）
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 5
输入起点:6
BFS Tree starting from vertex 北京:
|---北京
|------天津
|------郑州
|------呼和浩特
|---------沈阳
|---------徐州
|---------武汉
|---------西安
|---------兰州
|------------长春
|------------大连
|------------上海
|------------株洲
|------------成都
|------------西宁
|------------乌鲁木齐
|--------------哈尔滨
|--------------南昌
|--------------广州
|--------------贵阳
|--------------昆明
|----------------福州
|----------------深圳
|----------------柳州
|--------------------南宁
```

```
选择功能:
1. 深度优先搜索 (DFS)
2. 非递归深度优先搜索 DFS
3. 广度优先搜索 (BFS)
4. 深度优先生成树
5. 广度优先生成树
6. 打印图
0. Exit
输入功能: 6
Graph:

Vertex 乌鲁木齐: 兰州 (Weight: 1892)
Vertex 西宁: 兰州 (Weight: 216)
Vertex 兰州: 西安 (Weight: 676)  呼和浩特 (Weight: 1145)  西宁 (Weight: 216)  乌鲁木齐 (Weight: 1892)
Vertex 呼和浩特: 北京 (Weight: 668)  兰州 (Weight: 1145)
Vertex 西安: 成都 (Weight: 842)  郑州 (Weight: 511)  兰州 (Weight: 676)
Vertex 北京: 天津 (Weight: 137)  郑州 (Weight: 695)  呼和浩特 (Weight: 668)
Vertex 郑州: 武汉 (Weight: 534)  徐州 (Weight: 349)  北京 (Weight: 695)  西安 (Weight: 511)
Vertex 天津: 沈阳 (Weight: 704)  徐州 (Weight: 674)  北京 (Weight: 137)
Vertex 徐州: 上海 (Weight: 651)  天津 (Weight: 674)  郑州 (Weight: 349)
Vertex 沈阳: 长春 (Weight: 305)  大连 (Weight: 397)  天津 (Weight: 704)
Vertex 大连: 沈阳 (Weight: 397)
Vertex 长春: 哈尔滨 (Weight: 242)  沈阳 (Weight: 305)
Vertex 哈尔滨: 长春 (Weight: 242)
Vertex 成都: 贵阳 (Weight: 967)  昆明 (Weight: 1100)  西安 (Weight: 842)
Vertex 武汉: 株洲 (Weight: 409)  郑州 (Weight: 534)
Vertex 上海: 南昌 (Weight: 825)  徐州 (Weight: 651)
Vertex 昆明: 贵阳 (Weight: 639)  成都 (Weight: 1100)
Vertex 贵阳: 柳州 (Weight: 607)  株洲 (Weight: 902)  昆明 (Weight: 639)  成都 (Weight: 967)
Vertex 株洲: 广州 (Weight: 675)  南昌 (Weight: 367)  贵阳 (Weight: 902)  武汉 (Weight: 409)
Vertex 南昌: 福州 (Weight: 622)  株洲 (Weight: 367)  上海 (Weight: 825)
Vertex 福州: 南昌 (Weight: 622)
Vertex 南宁: 柳州 (Weight: 255)
Vertex 柳州: 南宁 (Weight: 255)  贵阳 (Weight: 607)
Vertex 广州: 深圳 (Weight: 140)  株洲 (Weight: 675)
Vertex 深圳: 广州 (Weight: 140)
请按任意键继续. . .
```

部分关键代码及其说明

定义了字典，输入边的格式，邻接多重表节点，图的顶点

```cpp
string name[26] = { "空",
    "乌鲁木齐","西宁","兰州","呼和浩特","西安",
    "北京","郑州","天津","徐州","沈阳",
    "大连","长春","哈尔滨","成都","武汉",
    "上海","昆明","贵阳","株洲","南昌",
    "福州","南宁","柳州","广州","深圳"
};

// 结构体表示边
struct Edge {
    int vertex1;  // 边的起点
    int vertex2;  // 边的终点
    int weight;   // 边的权重
};

// 邻接多重表的结点
struct ArcNode {
    int adjvex;               // 邻接结点的编号
    int weight;               // 边的权重
    ArcNode* nextarc;         // 指向下一个邻接结点
    ArcNode* opposite;        // 指向与当前边相反的边
};

// 图的顶点
struct VNode {
    int data;                 // 顶点编号
    ArcNode* firstarc;        // 指向第一个邻接结点
};
```

定义Graph类

```cpp
class Graph {
private:
    vector<VNode> vertices;  // 存储图的顶点
    stack<int> dfs_stack;     // 深度优先遍历辅助栈
    queue<int> bfs_queue;     // 广度优先遍历辅助队列
    queue<pair<int, int>> bfstree_queue;
    // 深度优先遍历递归函数
    void DFSRecursive1(int v, vector<bool>& visited);
    void DFSRecursive2(int v, vector<bool>& visited);

public:
    Graph(const vector<VNode>& initialVertices);

    void Print();
    // 添加无向边
    void AddEdge(const Edge& edge);

    // 深度优先遍历
    void DFS(int start);

    // 非递归深度优先遍历
    void DFSNonRecursive(int start);

    // 广度优先遍历
    void BFS1(int start);

    void BFS2(int start);

    void DFSRecursiveTreeIndent(int v, vector<bool>& visited, int depth);

    // 构建深度优先生成树
    void DFSTree(int start);

    // 构建广度优先生成树
    void BFSTree(int start);

    // 打印图的邻接多重表
    void PrintGraph();
};

```

向表中添加边

```cpp
void Graph::AddEdge(const Edge& edge) {
    ArcNode* arc1 = new ArcNode{ edge.vertex2, edge.weight, nullptr, nullptr };
    ArcNode* arc2 = new ArcNode{ edge.vertex1, edge.weight, nullptr, nullptr };

    // 添加边到v1的邻接多重表
    arc1->nextarc = vertices[edge.vertex1].firstarc;
    vertices[edge.vertex1].firstarc = arc1;

    // 添加边到v2的邻接多重表
    arc2->nextarc = vertices[edge.vertex2].firstarc;
    vertices[edge.vertex2].firstarc = arc2;

    // 设置边的相反边
    arc1->opposite = arc2;
    arc2->opposite = arc1;
}

```

递归实现深度优先搜索

C++

```cpp
void Graph::DFSRecursive1(int v, vector<bool>& visited) {
    // 标记当前结点为已访问
    visited[v] = true;
    cout << name[v] << " ";

    // 遍历邻接结点
    for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
        int adjvex = arc->adjvex;
        // 如果邻接结点未被访问，则递归访问
        if (!visited[adjvex]) {
            DFSRecursive1(adjvex, visited);
        }
    }
}

void Graph::DFSRecursive2(int v, vector<bool>& visited) {
    // 标记当前结点为已访问
    visited[v] = true;
    //cout << name[v] << " ";

    // 遍历邻接结点
    for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
        int adjvex = arc->adjvex;
        // 如果邻接结点未被访问，则递归访问
        if (!visited[adjvex]) {
            cout << "( " << v << " , " << adjvex << " ) ";
            DFSRecursive2(adjvex, visited);
        }
    }
}

void Graph::DFS(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);

    for (int i = start; i < vertices.size(); i++) {
        if(!visited[i]) DFSRecursive1(i, visited);
    }
    for (auto x : visited) {
        x = false;
    }
    cout << endl;
```

```
    for (int i = start; i < vertices.size(); i++) {
        if (!visited[i]) DFSRecursive2(i, visited);
    }
    cout << endl;
}

```

非递归实现深度优先搜索

```C++
void Graph::DFSNonRecursive(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);
    ArcNode* arc = vertices[start].firstarc;
    // 将起始结点入栈并输出
    dfs_stack.push(start);
    visited[start] = true;
    cout << name[start] << " ";

    while (!dfs_stack.empty()) {
        int v = dfs_stack.top();


        // 遍历邻接结点
        for (arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
            int adjvex = arc->adjvex;

            // 如果邻接结点未被访问，则输出并入栈
            if (!visited[adjvex]) {
                cout << name[adjvex] << " ";
                dfs_stack.push(adjvex);
                visited[adjvex] = true;
                break;
            }
        }
        if(arc==nullptr) dfs_stack.pop();
        //if(vertices[v].firstarc==nullptr) dfs_stack.pop();
    }

    cout << endl;
}

```

广度优先搜索

C++

```cpp
void Graph::BFS1(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);

    // 将起始结点入队列
    bfs_queue.push(start);

    while (!bfs_queue.empty()) {
        int v = bfs_queue.front();
        bfs_queue.pop();

        // 如果结点未被访问，则访问并将邻接结点入队列
        if (!visited[v]) {
            visited[v] = true;
            cout << name[v] << " ";

            // 遍历邻接结点
            for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
                int adjvex = arc->adjvex;
                if (!visited[adjvex]) {
                    bfs_queue.push(adjvex);
                }
            }
        }
    }

    cout << endl;
}

void Graph::BFS2(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);

    // 将起始结点入队列
    bfs_queue.push(start);

    while (!bfs_queue.empty()) {
        int v = bfs_queue.front();
        bfs_queue.pop();

        // 如果结点未被访问，则访问并将邻接结点入队列
        if (!visited[v]) {
            visited[v] = true;
```

```
44              /*cout << name[v] << " ";*/
45
46              // 遍历邻接结点
47              for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc
    = arc->nextarc) {
48                  int adjvex = arc->adjvex;
49                  if (!visited[adjvex]) {
50                      cout << "( " << v << " , " << adjvex << " ) ";
51                      bfs_queue.push(adjvex);
52                  }
53              }
54          }
55      }
56
57      cout << endl;
58  }
59
```

深度优先生成树

```cpp
void Graph::DFSRecursiveTreeIndent(int v, vector<bool>& visited, int depth) {
    // 标记结点为已访问
    visited[v] = true;
    cout << "|---";
    // 输出当前节点
    for (int i = 0; i < depth; ++i) {
        cout << "---";
    }
    cout << name[v] << endl;

    // 遍历邻接结点
    for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
        int adjvex = arc->adjvex;

        // 如果邻接结点未被访问，则递归输出并进行缩进
        if (!visited[adjvex]) {
            DFSRecursiveTreeIndent(adjvex, visited, depth + 1);
        }
    }
}

void Graph::DFSTree(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);

    // 递归构建深度优先生成树
    DFSRecursiveTreeIndent(start, visited, 0);
    cout << endl;
}

```

广度优先生成树

C++

```cpp
void Graph::BFSTree(int start) {
    // 初始化visited数组
    vector<bool> visited(vertices.size(), false);

    // 将起始结点入队列，并记录节点的深度
    bfstree_queue.push({ start, 0 });

    while (!bfstree_queue.empty()) {
        pair<int,int> front = bfstree_queue.front();
        bfstree_queue.pop();

        int v = front.first;
        int depth = front.second;

        // 如果结点未被访问，则访问并将邻接结点入队列
        if (!visited[v]) {
            visited[v] = true;

            cout << "|---";
            // 输出当前节点，根据深度确定缩进
            for (int i = 0; i < depth; ++i) {
                cout << "---";
            }
            cout << name[v] << endl;

            // 遍历邻接结点
            for (ArcNode* arc = vertices[v].firstarc; arc != nullptr; arc = arc->nextarc) {
                int adjvex = arc->adjvex;

                // 如果邻接结点未被访问，则入队列并记录深度
                if (!visited[adjvex]) {
                    bfstree_queue.push({ adjvex, depth + 1 });
                }
            }
        }
    }

    cout << endl;
}

```

邻接表形式打印图

```C++
void Graph::PrintGraph() {
    for (int i = 1; i < vertices.size(); ++i) {
        cout << "Vertex " << name[vertices[i].data] << ": ";
        for (ArcNode* arc = vertices[i].firstarc; arc != nullptr; arc = arc->nextarc) {
            cout << name[arc->adjvex] << " (Weight: " << arc->weight << ") ";
        }
        cout << endl;
    }
}
```

程序运行方式简要说明

main.cpp代码如下：

首先输出字典，然后创建对象，添加边，然后进入循环输入功能，执行相应的函数

main.cpp C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <set>
#include <windows.h>
#include "graph.h"
using namespace std;

int main() {
    system("color F0");
    cout << "字典:  " << endl;
    for (int i = 0; i < 26; i++) {
        cout << "点" << i << ": " << name[i] << endl;
    }

    // 输入边的数量
    int numEdges;
    cout << "Enter the number of edges(边数): ";
    cin >> numEdges;

    // 输入边的信息，并统计点的个数
    set<int> uniqueVertices;  // 使用集合来统计唯一的顶点编号
    vector<Edge> edges;

    cout << "以这个形式输入边 (vertex1 vertex2 weight):" << endl;
    for (int i = 0; i < numEdges; ++i) {
        Edge edge;
        cin >> edge.vertex1 >> edge.vertex2 >> edge.weight;

        // 统计顶点编号
        uniqueVertices.insert(edge.vertex1);
        uniqueVertices.insert(edge.vertex2);

        edges.push_back(edge);
    }

    // 初始化 vector<VNode> vertices
    vector<VNode> vertices(uniqueVertices.size() + 1);  // 从 1 开始编号

    // 为每个顶点赋值
    int index = 1;
    for (int vertex : uniqueVertices) {
        vertices[index].data = vertex;
```

```
45          ++index;
46      }
47
48      // 创建图并添加边
49      Graph graph(vertices);
50
51      for (const Edge& edge : edges) {
52          graph.AddEdge(edge);
53      }
54
55      while (true) {
56          // 打印菜单
57          system("cls");
58          graph.Print();
59          // 输入用户选择
60          int choice;
61
62          cin >> choice;
63          int n;
64          switch (choice) {
65          case 1:
66              cout << "输入起点:";
67              cin >> n;
68              cout << "DFS from vertex " << name[n] << " :" << endl;
69              graph.DFS(n);
70              system("pause");
71              system("cls");
72              break;
73          case 2:
74              cout << "输入起点:";
75              cin >> n;
76              cout << "Non-recursive DFS from vertex " << name[n] << ": " <
    < endl;
77              graph.DFSNonRecursive(n);
78              system("pause");
79              system("cls");
80              break;
81          case 3:
82              cout << "输入起点:";
83              cin >> n;
84              cout << "BFS starting from vertex " << name[n] << ": " << end
    l;
85              graph.BFS1(n);
86              cout << endl;
87              graph.BFS2(n);
88              system("pause");
89              system("cls");
```

```
90                 break;
91             case 4:
92                 cout << "输入起点:";
93                 cin >> n;
94                 cout << "DFS Tree starting from vertex " << name[n] << ": " <
    < endl;
95                 graph.DFSTree(n);
96                 system("pause");
97                 system("cls");
98                 break;
99             case 5:
100                cout << "输入起点:";
101                cin >> n;
102                cout << "BFS Tree starting from vertex " << name[n] << ": " <
    < endl;
103                graph.BFSTree(n);
104                system("pause");
105                system("cls");
106                break;
107            case 6:
108                cout << "Graph:" << endl << endl;
109                graph.PrintGraph();
110                system("pause");
111                system("cls");
112                break;
113            case 0:
114                cout << "Exiting program. Goodbye!" << endl;
115                return 0;
116            default:
117                cout << "Invalid choice. Please enter a valid option." << end
    l;
118            }
119        }
120
121        return 0;
122    }
```

附上Print函数：

C++

```cpp
void Graph::Print() {
    cout << "字典:  " << endl;
    for (int i = 0; i < 26; i++) {
        cout <<"点" << i << ": " << name[i] << endl;
    }

    cout << "\n选择功能:" << endl;
    cout << "1. 深度优先搜索 (DFS)" << endl;
    cout << "2. 非递归深度优先搜索 DFS" << endl;
    cout << "3. 广度优先搜索 (BFS)" << endl;
    cout << "4. 深度优先生成树" << endl;
    cout << "5. 广度优先生成树" << endl;
    cout << "6. 打印图" << endl;
    cout << "0. Exit" << endl;

    cout << "输入功能: ";

}

```