

Project6 实验报告

22336044 陈圳煌

1、程序功能简要说明

程序可以实现将图节点和边的数据通过邻接多重表和邻接表存储，实现基于邻接多重表的广度优先遍历和深度优先遍历（均采用非递归遍历），基于邻接表的非递归深度优先遍历和生成深度优先生成树、广度优先生成树。

2、部分代码以及程序说明

首先定义邻接多重表和邻接表的存储结构：

```
//邻接多重表
struct line{
    int ivex; //边的顶点1
    int jvex; //边的顶点2
    line *ilink; //连接顶点1的下一条边
    line *jlink; //连接顶点2的下一条边
    int weight;
};

struct graph1{
    int id;
    line* firstedge;
};

//邻接表
struct edge{
    int end;
    int weight;
    edge(int a,int b):end(a),weight(b){}
};

struct graph2{
    int id;
    vector<edge> next;
};
```

邻接表的存储：

```
//连接表的建立
edge tmp1(out,length);
edge tmp2(in,length);
s[in].next.push_back(tmp1);
s[out].next.push_back(tmp2);
```

其中 in 为第一个节点，out 为第二个节点，length 为连接两点的边上的权重。

邻接多重表的存储:

```
//多重连接表的建立
line* line1=new line;
line1->ivex=in;
line1->jvex=out;
line1->weight=length;
line1->ilink=NULL;
line1->jlink=NULL;
if(t[in].firstedge==NULL){
    t[in].firstedge=line1;
}else{
    line1->ilink=t[in].firstedge;
    t[in].firstedge=line1;
}

if(t[out].firstedge==NULL){
    t[out].firstedge=line1;
}else{
    line1->jlink=t[out].firstedge;
    t[out].firstedge=line1;
}
```

首先根据节点关系建立边，然后将边连接到相应节点的 firstedge 上，若已有数据，则使用前插法，实现 ilink 和 jlink 链表的连接。

深度优先遍历（邻接多重表）:

```
//深度优先遍历
void DFS(graph1 t[],int visit[],int begin,int n){
    stack<int> store; //使用栈实现非递归遍历
    vector<mymap> my; //用于存储边关系
    store.push(begin);
    int i=1; //用来与n比较确定是否遍历完
    cout << begin << " ";
    visit[begin]=1; //输出第一个节点
    while(!store.empty()){
        int id=store.top(); //取出栈顶元素并删除
        store.pop();
        line* e=t[id].firstedge; //定义栈顶元素对应的firstedge
        int next;
        int nid=id; //保存当前节点数据
        while(e!=NULL){
            next=(e->ivex==id)?e->jvex:e->ivex; //确定与当前节点相对的下一节点数据
            if(visit[next]==0){ //如果没有访问过
                cout << next << " "; //输出，访问
                i++;
                visit[next]=1;
                store.push(next); //这里插入两次，遍历过的节点不会再次遍历不影响结果，
                store.push(next); //但是可以处理遍历过的节点无法回退的问题
                mymap tmp(id,next,e->weight);
                int k=0;
                for(k=0;k<my.size();k++){
                    if(my[k].begin==id&&my[k].end==next){ //当该边为访问过则加入mymap
                        break;
                    }
                }
            }
            e=e->jlink;
        }
    }
}
```

```

67         if(k==my.size()){
68             my.push_back(tmp);
69         }
70         e=(e->ivex==id)?e->jlink:e->ilink;    //确定下一节点
71         id=next;
72     }else{                                   //后面节点遍历完成，横向寻找未遍历的节点
73         e=(e->ivex==nid)?e->ilink:e->jlink;
74     }
75 }
76 if(store.empty() && i<n){    //如果是非连通图，寻找为访问的节点
77     for(int j=0;j<n;j++){
78         if(visit[j]==0){
79             visit[j]=1;
80             cout << j << " ";
81             store.push(j);
82             break;
83         }
84     }
85 }
86
87 cout << endl << "边集为: " << endl;    //输出边集
88 for(int k=0;k<my.size();k++){
89     cout << "(" << my[k].begin << ", " << my[k].end << ") ->" << my[k].weight << endl;
90 }
91 }

```

广度优先遍历（邻接多重表）：

```

92 //广度优先遍历
93 void BFS(graphl t[],int visit[],int begin,int n){
94     queue<int> store;    //采用队列遍历
95     vector<mymap> my;
96     store.push(begin);
97     visit[begin]=1;
98     int i=0;
99     while(!store.empty()){
100         int id=store.front();    //取出队列头元素并删除
101         store.pop();
102         i++;    //遍历
103         cout << id << " ";
104         line* e=t[id].firstedge;    //以下思路与深度优先遍历类似
105         while(e!=NULL){
106             int next=((e->ivex==id)?e->jvex:e->ivex);
107             if(visit[next]==0){
108                 visit[next]=1;
109                 store.push(next);
110                 mymap tmp(id,next,e->weight);
111                 int k=0;
112                 for(k=0;k<my.size();k++){
113                     if(my[k].begin==id&&my[k].end==next){
114                         break;
115                     }
116                 }
117                 if(k==my.size()){
118                     my.push_back(tmp);
119                 }
120             }
121             e=((e->ivex==id)?e->ilink:e->jlink);    //区别：先进行横向遍历
122         }
123         if(store.empty() && i<n){
124             for(int j=0;j<n;j++){
125                 if(visit[j]==0){
126                     visit[j]=1;
127                     store.push(j);

```

```

127         store.push(j);
128         break;
129     }
130 }
131 }
132 }
133 cout << endl << "边集为: " << endl;
134 for(int k=0;k<my.size();k++){
135     cout << "(" << my[k].begin << ", " << my[k].end << ") ->" << my[k].weight << endl;
136 }
137 }

```

非递归深度遍历（邻接表）：

```

//非递归深度遍历
vector<int> dfscount; //存储每个节点的出度，方便后面建立深度优先生成树
void DFS_u(graph2 s[],int visit[],int n){
    stack<int> store;
    int id=0;
    int i=1;
    cout << id << " "; //输出第一个节点
    dfscount.push_back(s[0].next.size());
    visit[0]=1;
    for(id;i<n;i++){ //遍历
        if(s[id].next.size()){
            int j=s[id].next.size()-1;
            int count=0;
            for(j;j>=0;j--){ //先把同一层的节点存入栈中
                if(visit[s[id].next[j].end]==0){ //在与当前节点连接的节点中未遍历的成为出度
                    store.push(s[id].next[j].end);
                    count++;
                }
            }
            dfscount.push_back(count); //存储每个节点的出度
        }
        if(store.size()){ //存储同层节点后开始深度遍历
            id=store.top();
            if(visit[id]==0){
                visit[id]=1;
                i++;
                cout << id << " ";
            }
            store.pop();
        }
        if(store.size()==0&&i<n){ //处理非连通图的单独节点
            for(int k=0;k<n;k++){
                if(visit[k]==0){
                    store.push(k);
                    break;
                }
            }
        }
    }
}

```

建立深度优先生成树：

```

//深度优先生成树
void DFS_tree(graph2 s[],int visit[],int num,vector<int> &store_num){
    if(visit[num]==1){
        return;
    }
    visit[num]=1;
    store_num.push_back(num);
    for(int i=0;i<s[num].next.size();i++){ //递归算法进行深度优先遍历，将遍历完的序列存储，建立深度优先生成树
        if(visit[s[num].next[i].end]==0){
            DFS_tree(s,visit,s[num].next[i].end,store_num);
        }
    }
}

```

```

vector<int> store_num;
DFS_tree(s,visit,0,store_num);
for(int i=0;i<n;i++){
    visit[i]=0;
}
int sum=0;
int index=1;
//cout << store_num.size() << " " << dfscount.size();
for(int k=0;k<store_num.size();k++){
    cout << " " << store_num[k];
    for(int j=0;j<dfscount[k];j++){
        cout << "-";
    }
    index--;
    sum+=dfscount[k];
    if(index==0){
        cout << endl;
        index=sum;
        sum=0;
    }
}
cout << endl;



```

广度优先生成树:

```

91 //广度优先生成树
92 void BFS_tree(graph2 s[],int visit[],int num,int n){
93     queue<int> store; //使用队列
94     store.push(num);
95     vector<int> count;
96     vector<int> store_num;
97     visit[num]=1;
98     int i=0;
99     while(!store.empty()){ //非递归建立广度优先生成树
100         int top=store.front();
101         store_num.push_back(top);
102         i++;
103         store.pop();
104         int countnum=0;
105         for(int j=0;j<s[top].next.size();j++){ //将同层元素存入队列
106             if(visit[s[top].next[j].end]==0){
107                 visit[s[top].next[j].end]=1;
108                 store.push(s[top].next[j].end);
109                 countnum++; //存储未遍历的同层节点的个数
110             }
111         }
112         count.push_back(countnum);
113         if(i<n&&store.empty()){
114             for(int k=0;k<n;k++){
115                 if(visit[k]==0){
116                     store.push(k);
117                     visit[k]=1;
118                     break;
119                 }
120             }
121         }
122     }
123     int index=1;

```

译日志  调试  搜索结果

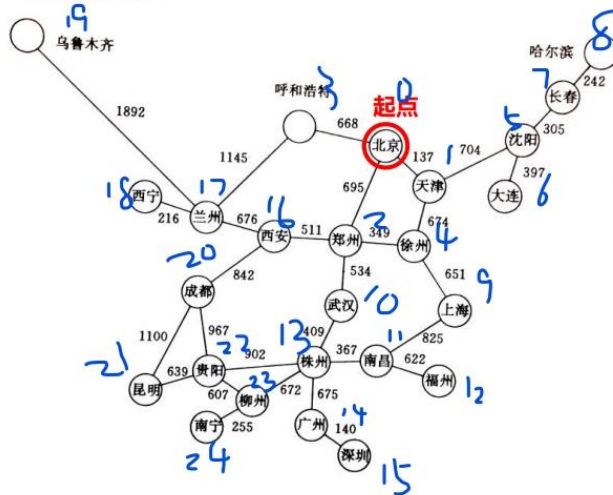
```

3      int index=1;
4      int sum=0;
5      for (int k=0;k<store_num.size();k++){
6          cout << " " <<store_num[k] ;
7          for (int j=0;j<count[k];j++){
8              cout << "-";          //当该节点的存在一个后继则输出" -"
9          }
10         index--;
11         sum+=count[k];
12         if (index==0){          //当该层根节点打印完进行换行操作
13             cout << endl;
14             index=sum;
15             sum=0;
16         }
17     }
18 }

```

3、测试

【测试数据】



以该图的边点关系为例，生成一个无向图。

测试数据

n=25;m=30

0 1 137; 0 2 695; 0 3 668; 1 4 674; 1 5 704; 5 6 397; 5 7 305; 7 8 242

4 9 651; 9 11 825; 11 12 622; 2 4 349; 2 10 534; 10 13 409; 11 13 367

2 16 511; 16 17 676; 3 17 1145; 17 19 1892; 17 18 216; 16 20 842

20 21 1100; 20 22 967; 21 22 639; 13 22 902; 22 23 607; 13 23 672

23 24 255; 13 14 675; 14 15 140

请选择你要进行的操作：

1. 深度优先遍历（邻接多重表）
2. 广度优先遍历（邻接多重表）
3. 深度优先遍历（非递归）
4. 深度优先生成树
5. 广度优先生成树
0. 退出

1

请输入开始节点：

0

0 3 17 16 2 10 13 14 15 23 22 11 12 9 4 1 5 7 8 6 21 20 24 18 19

边集为：

(0,3) ->668
(3,17) ->1145
(17,16) ->676
(16,2) ->511
(2,10) ->534
(10,13) ->409
(13,14) ->675
(14,15) ->140
(13,23) ->672
(23,22) ->607
(22,11) ->367
(11,12) ->622
(11,9) ->825
(9,4) ->651
(4,1) ->674

(1,5) ->704
(5,7) ->305
(7,8) ->242
(5,6) ->397
(22,21) ->639
(21,20) ->1100
(23,24) ->255
(17,18) ->216
(17,19) ->1892

请选择你要进行的操作：

1. 深度优先遍历（邻接多重表）
2. 广度优先遍历（邻接多重表）
3. 深度优先遍历（非递归）
4. 深度优先生成树
5. 广度优先生成树
0. 退出

2

请输入开始节点：

0

0 3 2 1 17 16 10 4 5 18 19 20 13 9 7 6 22 21 14 23 11 8 15 24 12

边集为：

(0,3) ->668
(0,2) ->695
(0,1) ->137
(3,17) ->1145
(2,16) ->511
(2,10) ->534
(2,4) ->349
(1,5) ->704
(17,18) ->216
(17,19) ->1892
(16,20) ->842
(10,13) ->409

```
(4,9) ->651
(5,7) ->305
(5,6) ->397
(20,22) ->967
(20,21) ->1100
(13,14) ->675
(13,23) ->672
(13,11) ->367
(7,8) ->242
(14,15) ->140
(23,24) ->255
(11,12) ->622
```

请选择你要进行的操作：

1. 深度优先遍历（邻接多重表）
2. 广度优先遍历（邻接多重表）
3. 深度优先遍历（非递归）
4. 深度优先生成树
5. 广度优先生成树
0. 退出

3

0 1 4 9 11 12 13 10 2 16 17 3 19 18 20 21 22 23 24 14 15 5 6 7 8

请选择你要进行的操作：

1. 深度优先遍历（邻接多重表）
2. 广度优先遍历（邻接多重表）
3. 深度优先遍历（非递归）
4. 深度优先生成树
5. 广度优先生成树
0. 退出

4

0---

1--- 4-- 9--

11- 12-- 13 10---- 2- 16- 17--

3--- 19 18 20 21-- 22- 23- 24- 14 15 5

6 7- 8

数字后的-表示有多少子树

0 后面---表示 0 之后有三个子树，根节点为 1，4，9；1 的子树为 11，12，

13；4 的子树为 10，2；9 的子树为 16，17；11 的子树 3；12 的子树 19，18；

13 没有子树，为叶子节点，后面的节点类似。


```

请选择你要进行的操作：
1.深度优先遍历（邻接多重表）
2.广度优先遍历（邻接多重表）
3.深度优先遍历（非递归）
4.深度优先生成树
5.广度优先生成树
0.退出
5
0---
1-- 2-- 3-
4- 5-- 10- 16- 17--
9- 6 7- 13--- 20- 19 18
11- 8 22 23- 14- 21
12 24 15

```

与深度优先生成树类似，数字后的-表示节点数。

4、程序运行方式

打开 exe 可执行文件。

```

请输入图的顶点数和边数：
25 30
请输入边之间的连接关系：（顶点1,顶点2,权重）

```

首先输入图的顶点数和边数。再输入边和节点和权重。

```

请选择你要进行的操作：
1.深度优先遍历（邻接多重表）
2.广度优先遍历（邻接多重表）
3.深度优先遍历（非递归）
4.深度优先生成树
5.广度优先生成树
0.退出

```

选择进行的操作，4 操作建立深度优先生成树需要先执行 3 非递归遍历获取每个节点的出度。