山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202000130143	姓名： 郑凯饶	班级： 计科 20.1
实验题目：图		
实验学时：2	实验日期：1215	

实验目的：

1. 掌握图的基本概念，图的描述方法；图上操作方法的实现。
2. 掌握图结构的应用。

软件开发环境：

Windows 10 家庭中文版 64 位（10.0，版本 18363）

Dev-C++ IDE

1. 实验内容

创建无向图类，存储结构使用邻接链表，提供操作：插入一条边，删除一条边，BFS,DFS。

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

```
class linkedGraph {
    protected:
        int n, e;
        list<int> *aList;
    public:
        linkedGraph(int n) {
            this -> n = n;
            e = 0;
            aList = new list<int> [n + 1];
        }

        ~linkedGraph() { delete [] aList; }

        int numberOfVertices() const { return n; }
        int numberOfEdges() const { return e; }
        bool directed() const { return false; }
        bool weighted() const { return false; }

        void insert(int, int);
        void erase(int, int);

        int degree(int) const;

        void bfs(int v, int vis[], int label);
        void dfs(int v, bool vis[]);

        int BFS(int s, int t);

        void sort() {
            for (int i = 1; i <= n; i ++) {
                aList[i].sort();
            }
        }
};
```

Protected 方法：

N,e:图的顶点数、边数

aList:声明一个大小为 n+1 的 list 数组

public 方法：

directed()及 weighted():该图为不含边权的无向图

insert()、erase()及 degree():调用 STL 中的 push_front(),remove()及 size()方法，进行点的邻接点的增删，从而实现边的增删

```
void linkedGraph::insert(int i, int j) {
//  aList[i].insert(aList[i].begin(), j);
//  aList[j].insert(aList[j].begin(), i);
    aList[i].push_front(j);
    aList[j].push_front(i);
    e ++;
}

void linkedGraph::erase(int i, int j) {
    aList[i].remove(j);
    aList[j].remove(i);
    e --;
}

int linkedGraph::degree(int v) const {
    return aList[v].size();
}
```

Sort()：由于题目要求的 dfs、bfs 序列满足字典序最小，调用 list 中的 sort()方法进行排序

Bfs()：对图进行 bfs 遍历，同时根据节点所在的连通块，为每个节点编号，最终可以得到图的连通分量数及每个节点所属的连通块编号

```
void linkedGraph::bfs(int v, int vis[], int label) {
    queue<int> q;
    vis[v] = label;
    q.push(v);

    while (!q.empty()) {
        int curNode = q.front();
        q.pop();

        if (label == -1) cout << curNode << " ";

        for (auto it = aList[curNode].begin(); it != aList[curNode].end(); it ++) {
            if (!vis[*it]) {
                q.push(*it);
                vis[*it] = label;
            }
        }
    }

}
```

Dfs()：深度优先搜索并输出

```
void linkedGraph::dfs(int v, bool vis[]) {
    vis[v] = 1;
    cout << v << " ";
    for (auto it = aList[v].begin(); it != aList[v].end(); it ++) {
        if (!vis[*it]) {
            dfs(*it, vis);
        }
    }
}
```

BFS()：与 bfs()方法不同，一旦搜索到 t 点则返回最短路径，piles 数组中存储着源点 s 到该点的最短路径并且 visit 数组保证每个节点只访问一次。已知，节点 A 由节点 B 出发第一次到达，应满足$piles[A] = piles[B] + 1$

```
int linkedGraph::BFS(int s, int t) {
    int *piles = new int[n + 1];
    bool *visit = new bool[n + 1];

    for (int i = 1; i <= n; i ++) {
        piles[i] = 0;
        visit[i] = 0;
    }

    queue<int> q;

    q.push(s);
    piles[s] = 0;
    visit[s] = 0;

    while (!q.empty()) {
        int f = q.front(); q.pop();
        for (auto it = aList[f].begin(); it != aList[f].end(); it ++) {
            if (!visit[*it]) {
                piles[*it] = piles[f] + 1;
                visit[*it] = 1;
                q.push(*it);

                if (*it == t) {
                    return piles[*it];
                }
            }
        }
    }
    return -1;
}
```

3. 测试结果（测试输入，测试输出）
在 OJ 平台上成功提交。

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）
这次的实验聚焦图的存储以及 DFS、BFS 方法。我采用 STL 中 list，list 中提供丰富的方法大大提高了我编码的效率。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
1. #include<bits/stdc++.h>
2. #include<list>
3. using namespace std;
4.
5. //list<int> lt;
6.
7. class linkedGraph {
8.  protected:
9.   int n, e;
10.  list<int> *aList;
11. public:
12.  linkedGraph(int n) {
13.   this -> n = n;
14.   e = 0;
15.   aList = new list<int> [n + 1];
16.  }
17.
18.  ~linkedGraph() { delete [] aList; }
19.
```

```
20.   int numberOfVertices() const { return n; }
21.   int numberOfEdges() const { return e; }
22.   bool directed() const { return false; }
23.   bool weighted() const { return false; }
24.
25.   void insert(int, int);
26.   void erase(int, int);
27.
28.   int degree(int) const;
29.
30.   void bfs(int v, int vis[], int label);
31.   void dfs(int v, bool vis[]);
32.
33.   int BFS(int s, int t);
34.
35.   void sort() {
36.    for (int i = 1; i <= n; i ++) {
37.     aList[i].sort();
38.    }
39.   }
40.};
41.
42.void linkedGraph::insert(int i, int j) {
43.// aList[i].insert(aList[i].begin(), j);
44.// aList[j].insert(aList[j].begin(), i);
45. aList[i].push_front(j);
46. aList[j].push_front(i);
47. e ++;
48.}
49.
50.void linkedGraph::erase(int i, int j) {
51. aList[i].remove(j);
52. aList[j].remove(i);
53. e --;
54.}
55.
56.int linkedGraph::degree(int v) const {
57. return aList[v].size();
58.}
59.
60.void linkedGraph::bfs(int v, int vis[], int label) {
61. queue<int> q;
62. vis[v] = label;
63. q.push(v);
64.
65. while (!q.empty()) {
```

```
  int curNode = q.front();
  q.pop();

  if (label == -1) cout << curNode << " ";

  for (auto it = aList[curNode].begin(); it != aList[curNode].end(); it ++) {
   if (!vis[*it]) {
    q.push(*it);
    vis[*it] = label;
   }
  }
 }

}

void linkedGraph::dfs(int v, bool vis[]) {
 vis[v] = 1;
 cout << v << " ";
 for (auto it = aList[v].begin(); it != aList[v].end(); it ++) {
  if (!vis[*it]) {
   dfs(*it, vis);
  }
 }
}

int linkedGraph::BFS(int s, int t) {
 int *piles = new int[n + 1];
 bool *visit = new bool[n + 1];

 for (int i = 1; i <= n; i ++) {
  piles[i] = 0;
  visit[i] = 0;
 }

 queue<int> q;

 q.push(s);
 piles[s] = 0;
 visit[s] = 0;

 while (!q.empty()) {
  int f = q.front(); q.pop();
  for (auto it = aList[f].begin(); it != aList[f].end(); it ++) {
   if (!visit[*it]) {
    piles[*it] = piles[f] + 1;
    visit[*it] = 1;
```

```
    q.push(*it);

    if (*it == t) {
     return piles[*it];
    }
   }
  }
 }
 return -1;
}

void solve() {
 int n, m, s, t;
 cin >> n >> m >> s >> t;

 linkedGraph G(n);

 for (int i = 1, op, u, v; i <= m; i ++) {
  cin >> op >> u >> v;

  if (op) {
   G.erase(u, v);
  }
  else {
   G.insert(u, v);
  }
 }

 int *c = new int[n + 1];
 for (int i = 1; i <= n; i ++) c[i] = 0;

 int label = 0;
 for (int i = 1; i <= n; i++) {
  if (!c[i]) {
   label ++;
   G.bfs(i, c, label);
  }
 }

 // 连通分量
 cout << label << '\n';

 // 每个连通子图中最小点编号
 int *over = new int[label + 1];
 for (int i = 1; i <= label; i ++) over[i] = 0;

```

```
158.  cout << 1 << " ";
159.  over[c[1]] ++;
160.
161.  for (int i = 2; i <= n; i++) {
162.   if (!over[c[i]]) {
163.    over[c[i]] = 1;
164.    cout << i << " ";
165.   }
166.   else over[c[i]] ++;
167.  }
168.  cout << '\n';
169.
170.  // 将每个节点的邻接点按字典序排序
171.  G.sort();
172.
173.
174.  // 从 s 点开始的 dfs 序列的长度
175.  cout << over[c[s]] << '\n';
176.
177.  // 从 s 点开始字典序最小的 dfs 序列
178.  bool *over_ = new bool[n + 1];
179.  for (int i = 1; i <= n; i ++) over_[i] = 0;
180.  G.dfs(s, over_);
181.  cout << '\n';
182.
183.  cout << over[c[t]] << '\n';
184.  int *over__ = new int[n + 1];
185.  for (int i = 1; i <= n; i ++) over__[i] = 0;
186.  G.bfs(t, over__, -1);
187.  cout << '\n';
188.
189.  // s 到 t 的最短路径 BFS
190.  cout <<  G.BFS(s, t) << '\n';
191. }
192.
193.
194. int main(){
195.  solve();
196.  return 0;
197. }
```