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：202200130048	姓名： 陈静雯	班级： 6
实验题目：图		
实验学时：4	实验日期： 12.14	

实验目的：
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。

软件开发工具：
Vscode

1. 实验内容
第一行四个整数 n，m，s，t。n（10≤n≤100000）代表图中点的个数，m（10≤m≤200000）代表接下来共有 m 个操作，s 代表起始点，t 代表终点。
接下来 m 行，每行代表一次插入或删除边的操作，操作格式为：
0 u v 在点 u 和 v 之间增加一条边
1 u v 删除点 u 和 v 之间的边

第一行输出图中有多少个连通分量
第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔
第三行输出从 s 点开始的 dfs 序列长度
第四行输出从 s 点开始的字典序最小的 dfs 序列
第五行输出从 t 点开始的 bfs 序列的长度
第六行输出从 t 点开始字典序最小的 bfs 序列
第七行输出从 s 点到 t 点的最短路径，若是不存在路径则输出-1

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）
邻接链表存某一节点的相连节点时从小到大存，即插入的时候找到合适的位置

连通分量：每一个未被访问的点都用 dfs 访问一遍，从点 1 开始，每访问一个点，联通分量个数+1，dfs 访问过的点更新其 reach 值，下次这个点就不会再次访问

升序输出最小点编号：即从点 1 开始进行 dfs，过程同上，访问一个点，输出这个点编号

Dfs 序列长度：s 点 dfs，即从 s 点跳到与它相邻的未访问过的节点，以此类推，每访问一个新的点，长度++

字典序最小的序列：因为链表是从小到大存的，所以直接 dfs 输出的就是字典序最小的

Bfs：从 s 点开始，把与 s 相邻的所有点都放入队列，下一次再从队首 pop 一个点，再把该点相邻的所有未被访问过的点 push 入队列，以此类推，每放入一个点（访问一个点），长度++

字典序最小的序列：同理，bfs 直接按访问的顺序输出即可

最短路径：bfs 的基础上，定义一个 reach2 数组，每一层的 reach2 值为上一层的值+1，因为每到一个点，都更新与它相连的所有点的路径，访问过的点不会访问第二遍，即第一次更新它的值即最短路径

3. 测试结果（测试输入，测试输出）

```
10 20 4 5
0 6 4
0 10 3
0 4 8
0 4 10
1 4 10
0 2 1
0 5 8
0 5 2
0 10 7
0 9 6
0 9 1
0 7 1
0 8 10
0 7 5
0 8 3
0 6 7
1 6 4
1 8 3
0 7 8
0 9 2
1
1
10
4 8 5 2 1 7 6 9 10 3
10
5 2 7 8 1 9 6 10 4 3
2
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

队列的空间要开够

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）

```
#include <iostream>
using namespace std;
```

```
int num=0;

template<class T>
class myqueue{
public:
    myqueue(int n=100);
    bool empty();
    void push(T& thelement);
    void pop();
    T front();
    T back();
    int size();
private:
    int queuefront;
    int queueback;
    int queuelength;
    int queuesize;
    T* element;
};

template<class T>
myqueue<T>::myqueue(int n){
    element=new T [n];
    queuefront=n-1;
    queueback=n-1;
    queuelength=n;
    queuesize=0;
}

template <class T>
bool myqueue<T>::empty(){
    return queuefront == queueback;
}

template <class T>
void myqueue<T>::push(T& thelement){
    if(queuefront==(queueback+1)%queuelength){                    //若空间不够,
重新进行动态分配
        queuelength*=2;
        T* temp = new T [queuelength];
        for(int i=0;i<queuelength;i++){
            temp[i]=element[i];
        }
        element=temp;
    }
    queueback = (queueback+1)%queuelength;
```

```
    element[queueback]=thelement;
    queuesize++;
}

template<class T>
void myqueue<T>::pop(){
    queuefront=(queuefront+1)%queuelength;
    element[queuefront]=0;
    queuesize--;
}

template<class T>
T myqueue<T>::front(){
    return element[(queuefront+1)%queuelength];
}

template<class T>
T myqueue<T>::back(){
    return element[queueback];
}

template<class T>
int myqueue<T>::size(){
    return queuesize;
}

template<class T>
struct chainnode{       //节点类
    T element;
    chainnode<T> *next;
    chainnode(){ }
    chainnode(const T& thelement){  //构造函数
        this->element=thelement;
    }
    chainnode(const T& element,chainnode<T>* next){
        this->element=element;
        this->next=next;
    }
};

template<class T>
class sortlist{                         //链表类
public:
    sortlist();
    void insert(const T& thelement);
    void erase(const T& thelement);
```

```
    ~sortlist(){
        while(firstnode!=NULL){
            chainnode<T>* temp=firstnode->next;
            delete firstnode;
            firstnode=temp;
        }
    }
    chainnode<T>* firstnode;
private:
    int size;
};

template<class T>
sortlist<T>::sortlist(){
    firstnode=NULL;
    size=0;
}

template<class T>
void sortlist<T>::insert(const T& thelement){
    size++;
    chainnode<T>* temp = firstnode;
    if(firstnode==NULL || thelement < temp->element){
        firstnode = new chainnode<T> (thelement);
        firstnode->next = temp;
        return ;
    }
    while(temp->next!=NULL && thelement>temp->next->element){
        temp=temp->next;
    }
    chainnode<T>* p = new chainnode<T> (thelement);
    p->next = temp->next;
    temp->next = p;
}

template<class T>
void sortlist<T>::erase(const T& thelement){    //删除节点
    if(firstnode==NULL) return ;
    chainnode<T>* temp = firstnode;
    size--;
    if(firstnode->element==thelement){     //删除的是头节点
        firstnode = firstnode->next;
        delete temp;
        return ;
    }
    while(temp->next!=NULL && temp->next->element!=thelement){
```

```
        temp=temp->next;
    }
    if(temp->next==NULL){                //没找到该节点
        return ;
    }
    chainnode<T>* p = temp->next;
    temp->next = p->next;
    delete p;
}

template<class T>
class linkgraph{
public:
    linkgraph(){ row = new sortlist<T> [100005]; }
    void insert(T u,T v);
    void erase(T u,T v);
    void bfs(int v,int reach[],int label);
    T length_bfs(int v,int reach[],int label);
    void dfs(int v,int reach[],int label);
    void n_dfs(int v,int reach[],int label);
    void length_dfs(int v,int reach[],int label);
    void minroad(int v,int reach[],int reach2[],int label);
private:
    sortlist<T>* row;
};

template<class T>
void linkgraph<T>::insert(T u,T v){
    row[u].insert(v);
    row[v].insert(u);
}

template<class T>
void linkgraph<T>::erase(T u,T v){
    row[u].erase(v);
    row[v].erase(u);
}

template<class T>
void linkgraph<T>::bfs(int v,int reach[],int label){
    myqueue<T>q(100005);
    reach[v]=label;
    q.push(v);
    while(!q.empty()){
        int w = q.front();
        cout<<w<<' ';
```

```
        q.pop();
        for(chainnode<int>* u = row[w].firstnode; u!=NULL; u = u->next){  //bfs
把一个节点连接的所有节点都放入队列，以此类推
            if(reach[u->element]==0){
                q.push(u->element);
                reach[u->element]=label;
            }
        }
    }
}

template<class T>
T linkgraph<T>::length_bfs(int v,int reach[],int label){
    int num=1;
    myqueue<T>q(100005);
    reach[v]=label;
    q.push(v);
    while(!q.empty()){
        int w = q.front();

        q.pop();
        for(chainnode<int>* u = row[w].firstnode; u!=NULL; u = u->next){
            if(reach[u->element]==0){
                q.push(u->element);
                num++;
                reach[u->element]=label;
            }
        }
    }
    return num;
}

template<class T>
void linkgraph<T>::dfs(int v,int reach[],int label){
    reach[v]=label;
    cout<<v<<' ';
    chainnode<T>* u = row[v].firstnode;
    while(u != NULL){
        if(reach[u->element]==0){
            dfs(u->element,reach,label);                    //不断访问相邻的节点直到
没有未被访问过的节点为止
        }
        u=u->next;
    }
}
```

```
template<class T>
void linkgraph<T>::n_dfs(int v,int reach[],int label){
    reach[v]=label;
    chainnode<T>* u = row[v].firstnode;
    while(u != NULL){
        if(reach[u->element]==0){
            n_dfs(u->element,reach,label);
        }
        u=u->next;
    }
}

template<class T>
void linkgraph<T>::length_dfs(int v,int reach[],int label){
    reach[v]=label;
    chainnode<T>* u = row[v].firstnode;
    while(u != NULL){
        if(reach[u->element]==0){
            num++;
            length_dfs(u->element,reach,label);
        }
        u=u->next;
    }
}

template<class T>
void linkgraph<T>::minroad(int v,int reach[],int reach2[],int label){
    myqueue<T>q(100005);
    reach[v]=label;
    q.push(v);
    while(!q.empty()){
        int w = q.front();
        q.pop();
        for(chainnode<int>* u = row[w].firstnode;u!=NULL; u = u->next){
            if(reach[u->element]==0){
                q.push(u->element);
                reach[u->element]=label;
                reach2[u->element]=reach2[w]+1;            //最小路即 bfs 的基础
上，定义一个 reach2，值为前一层元素的值+1
            }
        }
    }
}

int main(){
    int reach[100005];
```

```
    int reach2[100005];
    int n,m,s,t;
    cin>>n>>m>>s>>t;
    linkgraph<int> G;
    for(int i=0;i<m;i++){
        int p,u,v;
        cin>>p>>u>>v;
        if(p==0 && u!=v) G.insert(u,v);
        if(p==1 && u!=v) G.erase(u,v);
    }

    num=0;
    for(int i=0;i<=n;i++) reach[i]=0;
    for(int i=1;i<=n;i++){
        if(reach[i]==0){
            num++;
            G.n_dfs(i,reach,s);
        }
    }
    cout<<num<<endl;

    for(int i=0;i<=n;i++) reach[i]=0;
    for(int i=1;i<=n;i++){
        if(reach[i]==0){
            cout<<i<<' ';
            G.n_dfs(i,reach,s);
        }
    }
    cout<<endl;

    for(int i=0;i<=n;i++) reach[i]=0;
    num=1;
    G.length_dfs(s,reach,s);
    cout<<num<<endl;

    for(int i=0;i<=n;i++) reach[i]=0;
    G.dfs(s,reach,s);
    cout<<endl;

    for(int i=0;i<=n;i++) reach[i]=0;
    num = G.length_bfs(t,reach,t);
    cout<<num<<endl;

    for(int i=0;i<=n;i++) reach[i]=0;
    G.bfs(t,reach,t);
    cout<<endl;
```

```
    for(int i=0;i<=n;i++) reach[i]=0;
    for(int i=0;i<=n;i++) reach2[i]=0;
    G.minroad(s,reach,reach2,s);
    if(s==t) cout<<0<<endl;
    else if(reach2[t]==0) cout<<-1<<endl;
    else cout<<reach2[t]<<endl;
}
```