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　　：202200400053	姓名：　王宇涵	班级：　22 级 2 班
实验题目：图		
实验学时：2	实验日期：　2023-12-6	

实验目的：

1、掌握图的基本概念，图的描述方法；图上的操作方法实现。2、掌握图结构的应用。

软件开发环境：

VsCode

1. 实验内容

题目描述：

创建无向图类（采用邻接链表存储），提供操作：插入一条边、删除一条边、BFS、DFS。

输入输出格式：

输入：第一行输入四个整数 n、m、s、t，其中 n（10≤n≤100000）表示图中点的个数，m（10≤m≤200000）表示操作的次数，s、t 是图中的两个顶点。接下来 m 行，每行表示一次插入边或删除边操作：1.0 u v 在点 u、v 之间增加一条边； 2.1 u v 删除点 u、v 之间的边。

输出：依次输出如下 7 行： 第一行输出图中的连通分量个数； 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 点开始的字典序最小的 BFS 序列；第七行输出从 s 点到 t 点的最短路径长度，若不存在路径则输出－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）

Main 函数整体思路:

通过标准输入读取顶点数 n、边数 m、起点 s、终点 t 等信息。

使用输入的顶点数创建 LinkedGraph 对象 mGraph，该对象表示整个图结构。

使用循环读取输入，根据操作类型 op 和边的起点 u 和终点 v，进行图的操作。

当 op 为 0 时，调用 mGraph.insertEdge(new UnWeightedEdge(u, v)) 向图中插入一条边。

当 op 不为 0 时，调用 mGraph.eraseEdge(u, v) 从图中删除一条边。

调用 mGraph.lableComponents(c) 对图的连通分量进行标记，结果存储在数组 c 中,输出标记个数.

调用 mGraph.ldfs(s, c) 和 mGraph.rdfs(s, c) 分别进行深度优先搜索（DFS），输出遍历序列长度和遍历序列。

调用 mGraph.lbfs(t, c) 和 mGraph.rbfs(t, c) 分别进行广度优先搜索（BFS），输出遍历序列长度和遍历序列。

调用 mGraph.findMinPath(s, t, path, c) 寻找起点 s 到终点 t 的最短路径，并输出最短路径的长度。

方法解析

insertEdge 方法：用于向图中添加一条边。对于无向图，它同时插入两个顶点对应的链表。注意由于需要字典序最小的 dfs 和 bfs 序列,所以每次插入都要保证链表是有序的.

eraseEdge 方法：从图中删除一条边。对于无向图，它同时删除两个顶点对应的链表。

lableComponents 方法：这个方法用于标记图的连通分量。它采用了广度优先搜索（BFS）的思想，从某个尚未标记的顶点开始，通过 BFS 将所有连通的顶点标记为同一个连通分量。重复这个过程，直到所有的顶点都被标记,最后返回的值 lable 就是标记的连通分量个数.

ldfs 和 rdfs 方法：这两个方法分别进行深度优先搜索（DFS），从给定的顶点出发，访问和标记其可达的所有顶点。ldfs 方法在搜索过程中记录并返回 DFS 遍历序列的长度,而 rdfs 方法只输出遍历序列，不返回长度。

lbfs 和 rbfs 方法：这两个方法分别进行广度优先搜索（BFS），从给定的顶点出发，访问和标记其可达的所有顶点。lbfs 方法在搜索过程中记录并返回 BFS 遍历序列的长度,而 rbfs 方法只输出遍历序列。

findMinPath 方法：用于查找两个顶点之间的最短路径。它采用 BFS 的思路，从起点开始进行 BFS 搜索，同时记录路径长度，并在找到终点时返回最短路径的长度。这个方法也是通过 BFS 进行实现，但在搜索过程中记录了路径长度，并在找到终点时立即返回最短路径长度。

- 注意:以上所有的 dfs 和 bfs 都需要初始化标记数组 reach[]为 0.

3. **测试结果（测试输入，测试输出）**

输入

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

输出

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）

本次实验也存在一些难点,主要是对于 bfs 和 dfs 的理解以及代码量的庞大.

问题一:如何实现无向图?

答: 继承于有向图, 插入和删除的时候同时插入和删除两条边,但是注意边数的变化值为 1

问题二:如何输出字典序最小的 dfs(bfs)序列?

答: 存储边的时候就使得每个结点对应的链表按照顺序存储即可,需要更改 insert 函数.

问题三:如何求出无权的最短路径?

答: 由课堂知识可得,dfs 无法实现要求,bfs 可以实现要求,只要达到结点就返回存储的路径长度即可.

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）

```
#pragma once
```

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

template <class T>
class Edge
{
    public:
        virtual ~Edge() {};
        virtual int vertex1() const = 0;
        virtual int vertex2() const = 0;
        virtual T weight() const = 0;
};



using namespace std;

class UnWeightedEdge: public Edge<bool>
{
      public:
            int vertex1() const
            {
                  return v1;
            }
            int vertex2() const
            {
                  return v2;
            }
            UnWeightedEdge(int v1,int v2)
            {
                  this->v1=v1;
```

```
            this->v2=v2;
        }
        bool weight()const
        {
            return false;
        }

    protected:
        int v1,v2;
};
#pragma once


#pragma once
#pragma once
#include<iostream>
using namespace std;
#pragma once
#include<iostream>
using namespace std;


template<class T>
class VertexIterator
{
    public:
        virtual ~VertexIterator(){}
        virtual int next()=0;
        virtual int next(T& )=0;
};
#include<queue>
template<class T>//权的类型
class graph
{
```

```
public:
    virtual ~graph(){};

    virtual int numberOfVertices()const=0;
    virtual int numberOfEdges()const =0;
    virtual bool existsEdge(int ,int )const =0;
    virtual void insertEdge(Edge<T>* )=0;
    virtual void eraseEdge(int ,int)=0;
    virtual int inDegree(int) const =0;
    virtual int outDegree(int )const =0;

    virtual bool directed()const =0;
    virtual bool weighted()const =0;
    virtual VertexIterator<T>* iterator(int )=0;

    virtual void bfs(int v,int reach[],int lable)
    {
        queue<int>q ;
        reach[v]=lable;
        q.push(v);

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

            VertexIterator<T>* iw =iterator(w);

            int u;
            while((u=iw->next())!=0)
            {
                if(reach[u]==0)
```

```
                {
                    q.push(u);
                    reach[u]=lable;
                }
            }
            delete iw;
        }
    }
    //求长度
    virtual int lbfs(int v,int reach[])
    {
        length=0;
        this->reach=reach;
        l2bfs(v);
        return length;
    }

    virtual void l2bfs(int v)
    {
        reach[v]=1;
        queue<int>q ;
        q.push(v);

        while(!q.empty())
        {
            length++;
            int w=q.front();
            q.pop();
            VertexIterator<T>* iw =iterator(w);
            int u;
            while((u=iw->next())!=0)
            {
```

```
                    if(reach[u]==0)
                    {
                        q.push(u);
                        reach[u]=1;
                    }
                }
                delete iw;
            }
        }

        //求序列
        virtual void rbfs(int v,int reach[])
        {
            this->reach=reach;
            r2bfs(v);
        }
        virtual void r2bfs(int v)
        {
            reach[v]=1;
            queue<int>q ;
            q.push(v);
            while(!q.empty())
            {
                int w=q.front();
                q.pop();

                cout<<w<<" ";
                VertexIterator<T>* iw =iterator(w);

                int u;
                while((u=iw->next())!=0)
                {
```

```
                    if(reach[u]==0)
                    {
                        q.push(u);
                        reach[u]=1;
                    }
                }
                delete iw;
            }
        }
        //求长度
        int    ldfs(int v,int reach[])
        {
            length=1;
            this->reach=reach;
            l2dfs(v);
            return length;
        }

        void    l2dfs(int v)
        {
            reach[v]=lable;
            VertexIterator<T>* iv=iterator(v);
            int u;
            while((u=iv->next())!=0)
            {
                //没有被遍历过
                if(reach[u]==0)
                {
                    l2dfs(u);
                    length++;
                }
            }
```

```
        delete iv;
    }

    void rdfs(int v,int reach[])
    {
        this->reach=reach;
        r2dfs(v);
    }

    void r2dfs(int v)
    {
        cout<<v<<" ";
        reach[v]=lable;
        VertexIterator<T>* iv=iterator(v);
        int u;
        while((u=iv->next())!=0)
        {
            //没有被遍历过
            if(reach[u]==0)
                r2dfs(u);
        }
        delete iv;
    }
    //找最短路径
    int findMinPath(int s,int t,int path[],int reach[])
    {
        int n=numberOfVertices();
        this->path=path;
        this->reach=reach;

        length=0;
```

```
queue<int>q;
reach[s]=1;
q.push(s);

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

    VertexIterator<bool>* is= iterator(w);

    int u;
    while((u=is->next())!=0)
    {
        //如果还没有遍历
        if(reach[u]==0)
        {
            //找到了终点
            if(u==t)
            {
                return path[w]+1;
            }
            //没到终点,更新从 w->u 的路
            else
            {
                path[u]=path[w]+1;
                reach[u]=1;
                q.push(u);
            }
        }
    }
```

```
            }
            return -1;
        }

        int lableComponents(int c[])
        {
            int n=numberOfVertices();
            lable=0;
            //遍历所有元素
            for(int i=1;i<=n;i++)
            //如果没有被标记
            if(c[i]==0)
            {
                lable++;
                bfs(i,c,lable);
            }
            return lable;
        }

        int desitination;
        int *path;
        int length;
        int* reach;
        int lable;
};
#pragma once
#pragma once
template<class T>
class chainNode
{
    public:
    T element;
```

```
        chainNode<T>* next;


        chainNode(){};


        //两个构造函数
        chainNode(const T& element){this->element=element;}
        chainNode(const T& element,chainNode<T>*next)
        {
            this->element=element;
            this->next=next;
        }
};
#pragma once
#include<iostream>
using namespace std;
template <class T>
class LinearList
{
        public:
        virtual ~LinearList(){};
        virtual bool empty()const =0;
        virtual int size()const =0;
        virtual T get(int theIndex)const =0;
        virtual int indexOf(const T& x)const =0;
        virtual void erase(int theIndex)=0;
        virtual void insert(int theIndex,const T &x)=0;
        virtual void output(ostream& out)const=0;
        virtual void clear()=0;
        virtual void push_back(const T& x)=0;
};
#include<sstream>
template<class T>
```

```
class GraphChain:public LinearList<T>
{
public:
    //似乎没用
    GraphChain()
    {
        firstNode=lastNode=NULL;
        listSize=0;
    }
    GraphChain(int initialCapacity);
    GraphChain(const GraphChain<T>&);
    ~GraphChain();

    //ADT
    bool empty()const    {return listSize==0;}
    int size()const {return listSize;}
    T get(int theIndex)const;
    int indexOf(const T& x)const;
    void erase(int theIndex);
    void insert(int theIndex,const T &x);
    void insertVertex(int theVertex);
    void output(ostream& out)const;
    void checkIndex(int theIndex)const;
    void clear();
    void push_back(const T& x);
    void set(int theIndex,T x);
    void reverse();
    //新增 ADT
    T*    eraseElement(int theVertex);
public:
    chainNode<T>* firstNode;
    chainNode<T>* lastNode;
```

```
    int listSize;


};
//赋值构造函数
template<class T>
GraphChain<T>::GraphChain(int initialCapacity)
{
    if(initialCapacity<1)return;
    firstNode=lastNode=NULL;
    listSize=0;
}
//拷贝构造函数
template<class T>
GraphChain<T>::GraphChain(const GraphChain<T>&theList)
{
    listSize=theList.listSize;
    if(listSize==0)
    {
        firstNode=lastNode=NULL;
        return;
    }
    else
    {
        //先把被 copy 链表第一个结点作为第一个结点
        //sourceNode:指向被 copy 的结点
        //targetNode:指向 copy 链表的尾部结点
        chainNode<T>* sourceNode=theList.firstNode;
        firstNode=new chainNode<T>(sourceNode->element);
        sourceNode=sourceNode->next;
        chainNode<T>* targetNode=firstNode;
        while(sourceNode!=NULL)
        {
```

```
                targetNode->next=new chainNode<T>(sourceNode->element);
                targetNode=targetNode->next;
                sourceNode=sourceNode->next;
            }
            targetNode->next=NULL;
            lastNode=targetNode;
        }
}

//检查索引
template<class T>
void GraphChain<T>::checkIndex(int theIndex)const
{
    if(theIndex<0||theIndex>=listSize)
    {
        ostringstream s;
        s<<"index="<<theIndex<<"size="<<listSize;
        throw(s.str());
    }
}
//析构函数
template<class T>
GraphChain<T>::~GraphChain()
{
    while(firstNode!=NULL)
    {
        chainNode<T>* tmp=firstNode->next;
        delete firstNode;
        firstNode=tmp;
    }
}
```

```
//得到元素
template<class T>
T GraphChain<T>::get(int theIndex)const
{
	checkIndex(theIndex);
	chainNode<T>* currentNode=firstNode;
	for(int i=0;i<theIndex;i++)
	{
		currentNode=currentNode->next;
	}
	return currentNode->element;
}

//查询第一次出现的索引
template<class T>
int GraphChain<T>::indexOf(const T& x)const
{
	chainNode<T>* currentNode=firstNode;
	int index=0;
	while(currentNode!=NULL&&currentNode->element!=x)
	{
		currentNode=currentNode->next;
		index++;
	}
	if(currentNode==NULL)
	return -1;
	else return index;
}

//删除结点
template<class T>
void GraphChain<T>::erase(int theIndex)
```

```
{
    checkIndex(theIndex);
    chainNode<T>* deleteNode;
    //先判断是否删除第 0 结点
    if(theIndex==0)
    {
        deleteNode=firstNode;
        firstNode=firstNode->next;
    }
    //找到第 theIndex-1 的位置 p
    else
    {
        chainNode<T>*p=firstNode;
        for(int i=0;i<theIndex-1;i++)
        p=p->next;

        deleteNode=p->next;
        p->next=deleteNode->next;
    }
    delete deleteNode;
    listSize--;
    chainNode<T>* p=firstNode;
    for(int i=0;i<listSize-1;i++)
    p=p->next;
    lastNode=p;
}

//插入结点
template<class T>
void GraphChain<T>::insert(int theIndex,const T&x)
{
    if(theIndex<0||theIndex>listSize)
```

```
        {
            ostringstream s;
            s<<"index="<<theIndex<<"size="<<listSize;
            throw(s.str());
        }

        //无需扩容
        if(theIndex==0)
        firstNode=new chainNode<T>(x,firstNode);
        else
        {
            chainNode<T>*p=firstNode;
            for(int i=0;i<theIndex-1;i++)
            p=p->next;
            p->next=new chainNode<T>(x,p->next);
        }
        listSize++;
        chainNode<T>* p=firstNode;
        for(int i=0;i<listSize-1;i++)
        p=p->next;
        lastNode=p;
}

template<class T>
inline void GraphChain<T>::insertVertex(int theVertex)
{
        //找到 theVertex 应该待的位置
        chainNode<T>* p= firstNode;
        chainNode<T>* pp=NULL;
        chainNode<T>* newNode=new chainNode<T>(theVertex,NULL);
        //如果链表是空的
        if(p==NULL)
```

```
    {
        firstNode=newNode;
    }
    else
    {
        //找到第一个大于 thevertex 的
        while(p!=NULL&&p->element<theVertex)
        {
            pp=p;
            p=p->next;
        }
        //没有找到,此时 pp 指向最后一个元素,p 指向 null
        if(p==NULL)
        {
            pp->next=newNode;
        }
        //如果需要头插
        else if(p==firstNode)
        {
            newNode->next=firstNode;
            firstNode=newNode;
        }
        //如果不需要头插,此时 pp 指向小于 thevertex 的元素,p 指向大于 thevertex 的元素
        else
        {
            newNode->next=p;
            pp->next=newNode;
        }
    }
    listSize++;
    return;
}
```

```
//输出
template<class T>
void GraphChain<T>::output(ostream& out)const
{
	if(listSize==0)
	{
		out<<"empty";return;
	}

	for(chainNode<T>*currentNode=firstNode;currentNode!=NULL;currentNode=currentNode->next)
	{
		out<<currentNode->element<<" ";
	}
}
//重载
template<class T>
ostream& operator<<(ostream& out,const GraphChain<T>x)
{
	x.output(out);return out;
}

//后缀元素
template<class T>
void GraphChain<T>::clear()
{
	while(firstNode!=NULL)
	{
		chainNode<T>* nextNode=firstNode->next;
		delete firstNode;
		firstNode=nextNode;
	}
```

```
    listSize=0;
}

template<class T>
void GraphChain<T>::push_back(const T&x)
{
    chainNode<T>* newNode=new chainNode<T>(x,NULL);
    if(firstNode==NULL)
        firstNode=lastNode=newNode;
    else
    {
        lastNode->next=newNode;
        lastNode=newNode;
    }
    listSize++;
}

template<class T>
void GraphChain<T>::set(int theIndex,T x)
{
    checkIndex(theIndex);
    chainNode<T>* currentNode=firstNode;
    for(int i=0;i<theIndex;i++)
        currentNode=currentNode->next;
    currentNode->element=x;
    return ;
}

template<class T>
void GraphChain<T>::reverse()
{
    chainNode<T>* currentNode=firstNode;
```

```
        chainNode<T>* previousNode=NULL;
        chainNode<T>*nextNode=firstNode;
        lastNode=firstNode;
        while(currentNode!=NULL)
        {
                nextNode=currentNode->next;
                currentNode->next=previousNode;
                previousNode=currentNode;
                currentNode=nextNode;
        }
        firstNode=previousNode;
}

template <class T>
inline T*    GraphChain<T>::eraseElement(int theVertex)
{
        chainNode<T> * p= firstNode;
        chainNode<T> *pp=NULL;
        chainNode<T> *returnNode=NULL;
        while(p!=NULL&&p->element!=theVertex)
        {
                pp=p;
                p=p->next;
        }


        if(p==NULL)
                return NULL;

        returnNode=p;

        if(p==firstNode)
```

```
        {
            firstNode=p->next;
        }
        else
        {
            pp->next=p->next;
        }

        listSize--;
        return &returnNode->element;
}



class LinkedDigraph:public graph<bool>
{
    protected:
        int n;
        int e;
        GraphChain<int> *aList; //邻接表

    public:
        LinkedDigraph(int numberOfVertices=0)
        {
            if(numberOfVertices<0)
            {
                cout<<"wrong"<<endl;
                return;
            }
            n=numberOfVertices;
            e=0;
            aList= new GraphChain<int> [n+1];
        }
```

```
~LinkedDigraph(){delete[] aList;}

int numberOfVertices()const
{
    return n;
}

int numberOfEdges()const
{
    return e;
}

bool existsEdge(int i,int j)const
{
    if(i>=1&&i<=n&&j>=1&&j<=n&&aList[i].indexOf(j)!=-1)
        return true;
    else
        return false;
}

void insertEdge(Edge<bool>* theEdge)
{
    int v1=theEdge->vertex1();
    int v2=theEdge->vertex2();
    //新边才插入
    if(aList[v1].indexOf(v2)==-1)
    {
        e++;
        aList[v1].insertVertex(v2);
    }
}
```

```
void eraseEdge(int i,int j)
{

    if(i>=1&&i<=n&&j>=1&&j<=n)
    {
        int *v=aList[i].eraseElement(j);
        if(v!=NULL)
            e--;
    }
}

int inDegree(int theVertex) const
{
    int sum=0;
    for(int i=1;i<=n;i++)
    {
        if(aList[i].indexOf(theVertex)!=-1)
            sum++;
    }
    return sum;
}
int outDegree(int theVertex)const
{
    if(theVertex>=1&&theVertex<=n)
        return aList[theVertex].size();
    else
        return 0;
}
bool directed()const
{
    return true;
```

```
    }
    bool weighted()const
    {
        return false;
    }
    class myIterator : public VertexIterator<bool>
{
     public:
        myIterator(chainNode<int> *theNode)
        {
            currentNode = theNode;
        }

        ~myIterator() {}

        int next()
        {
            if (currentNode == NULL)
                return 0;
            int nextVertex = currentNode->element;
            currentNode = currentNode->next;
            return nextVertex;
        }

        int next(bool& theWeight)
        {
            if (currentNode == NULL)
                return 0;
            int nextVertex = currentNode->element;
            currentNode = currentNode->next;
            theWeight = true;
            return nextVertex;
```

```
            }

        protected:
            chainNode<int> *currentNode;
    };

    myIterator* iterator(int theVertex)
    {// Return iterator for vertex theVertex.
        if(theVertex>=1&&theVertex<=n)
            return new myIterator(aList[theVertex].firstNode);
        else
            return NULL;
    }
};

class LinkedGraph:public LinkedDigraph
{
    public:
        using LinkedDigraph::aList;
        LinkedGraph(int numberOfVertices=0):LinkedDigraph(numberOfVertices){}

        bool directed()const {return false;}

        void insertEdge(Edge<bool>* theEdge)
        {
            int pre=e;

            LinkedDigraph::insertEdge(theEdge);
            //如果增加边了,则增加相同的边
            if(e>pre)
            {
                LinkedDigraph::insertEdge(new UnWeightedEdge(theEdge->vertex2(),
```

```
                    theEdge->vertex1()));
                    e--;
                }
            }
            void eraseEdge(int i,int j)
            {
                int pre=e;
                LinkedDigraph::eraseEdge(i,j);
                //如果减少边了,那么就再减少
                if(e<pre)
                {
                    LinkedDigraph::eraseEdge(j,i);
                    e++;
                }
            }
};
#include<cstring>
int n,m,s,t;
const int N=100010;

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin>>n>>m>>s>>t;
    LinkedGraph mGraph(n);
    int op,u,v;
    while(m--)
    {
        cin>>op>>u>>v;
        //增加一条边
        if(op==0)
```

```
        {
            mGraph.insertEdge(new UnWeightedEdge(u,v));
        }
        else
        {
            mGraph.eraseEdge(u,v);
        }
    }

    int c[n+1]={0};
    //输出连通分量
    cout<<mGraph.lableComponents(c)<<endl;

    //输出所有连通子图最小点的标号
    bool st[n+1]={false};
    for(int i=1;i<=n;i++)
    {
        if(st[c[i]]==false)
        {
            cout<<i<<" ";
            st[c[i]]=true;
        }
    }
    cout<<endl;

    //输出从 s 点开始的 dfs 序列长度
    for(int i=0;i<=n;i++)
        c[i]=0;
    cout<<mGraph.ldfs(s,c)<<endl;

    //输出从 s 点开始的 dfs 序列
    for(int i=0;i<=n;i++)
```

```
        c[i]=0;
    mGraph.rdfs(s,c);
    cout<<endl;

    //输出从 t 点开始的 bfs 序列长度
     for(int i=0;i<=n;i++)
        c[i]=0;
    cout<<mGraph.lbfs(t,c)<<endl;

    //输出从 t 点开始的 bfs 序列

    for(int i=0;i<=n;i++)
        c[i]=0;
    mGraph.rbfs(t,c);
    cout<<endl;

    //输出最短路径
    for(int i=0;i<=n;i++)
        c[i]=0;
    int path[n+1]={0};

    cout<<mGraph.findMinPath(s,t,path,c);
}
```