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：202000130198	姓名： 隋春雨	班级： 20.4
实验题目：图		
实验学时：2	实验日期： 2021-12-16	

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：

CLION2020

1. 实验内容

题目描述：

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。

输入输出格式：

输入：

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。

输出：

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）

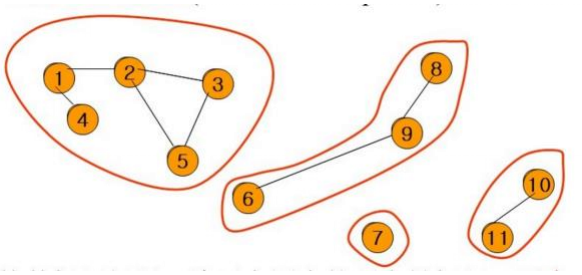
(一)数据结构：图和临接链表

1. 临接链表：我们维护一个有序链表。通过这个链表，访问其结点，使用有序链表能够提高时间性能。

2. 图：使用图这个数据结构，进行 dfs、bfs、查询联通分量等操作。图内的数据成员有 chain 数组和结点数量，提供的操作如下：

```
    void insert(int i, int j);//插入
    void erase(int i, int j);//删除
    pair<int, vector<int>> Connected_Component()const;//联通分量，返回个数和联通集
    pair<int, vector<int>>dfs(int s);//返回dfs序列长度和字典序最小的dfs序列
    pair<int, vector<int>>bfs(int s)const;//返回bfs序列和字典序最小的dfs序列
    int distance(int i, int j)const;//返回i和j之间的长度
```

图的实例：



(二)算法：

1. 计算连通分量：我们遍历每一个结点，并维护一个 jud 数组，对每一个未被访问的结点进行 bfs 访问。每一次在 for 循环里压入一个新的结点的时候，cnt 自增，表示有一个新的连通分量。同时，对这个结点所在的联通图进行搜寻，把与其临接的每个结点都标记为不可访问即可。代码如下：

```
template<class T>
pair<int, vector<int>> Graph<T>::Connected_Component() const
{
    int cnt = 0;
    queue<int>q;
    vector<int>sequence;
    bool* jud = new bool[1 + sizeofNode]();//初始化
    for (int i = 1; i <= sizeofNode; i++)
    {
        if (!jud[i])//没有被访问过
        {
            sequence.push_back(i);
            cnt++;
            q.push(i);
            jud[i] = true;
            while (!q.empty())
            {
                int front = q.front();
                q.pop();
                //将临接的所有顶点压入
```

```
                if (!jud[p->element])
                {
                    q.push(p->element);
                    jud[p->element] = true;
                }
            }
        }
    }

    return pair<int, vector<int>>(cnt, sequence);
}
```

保存每个联通分量的最小点：在 for 循环中进行保存即可。

2. Dfs 操作：我们使用一个共有接口函数分别调用两个操作。代码如下：

```
template<class T>
pair<int, vector<int>> Graph<T>::dfs(int s)
{
    bool* jud = new bool[sizeofNode + 1]();//初始化
    jud[s] = true;
    int length = dfs_length(s, jud);//计算Length
    for (int i = 0; i <= sizeofNode; i++)
    {
        jud[i] = false;//标记为false
    }
    vector<int>ans;
    ans.push_back(s);
    jud[s] = true;
    dfs_sequence(s, &ans, jud);//保存序列
    return pair<int, vector<int>>(length, ans);
}
```

3. 输出 dfs 序列：对输入的变量的临接的每一个未相邻的结点进行访问并进行 dfs，同时使用一个数组进行保存。代码如下：

```
template<class T>
void Graph<T>::dfs_sequence(int s, vector<int>& v, bool* jud) const
{
    for (chainNode<T>* p = ptr[s].getFirstNode(); p; p = p->next)
    {
        if (!jud[p->element])
        {
            jud[p->element] = true;
            v.push_back(p->element);
            dfs_sequence(p->element, &v, jud);//递归调用
        }
    }
}
```

4. 计算两点之间距离：由于这个是无权重图，所以我们可以使用 bfs 直接进行计算。我们使用一个队列来进行完成该函数，每次从队列中取出来队头，判断一下是否为我们想要到达的顶点，如果是，直接退出。否则，遍历临接的每一个没有被访问过的点，压入队列。若最后退出的时候也没有访问到，返回-1

代码如下：

```
245     template<class T>
246     int Graph<T>::distance(int i, int j) const//返回i和j之间的距离
247     {
248         int length = 0;
249         bool* jud = new bool[sizeofNode + 1]();
250         queue<int>q;
251         q.push(i);//压入
252         jud[i] = true;
253         while (!q.empty())
254         {
255             int front = q.front();
256             q.pop();
257             if (front == j)//找到了
258             {
259                 return length;
260             }
261             else
262             {
```

```
261             else
262             {
263                 length++;
264                 for (chainNode<T>* p = ptr[front].getFirstNode(); p; p = p->next)
265                 {
266                     if (!jud[p->element])
267                     {
268                         q.push(p->element);
269                         jud[p->element] = true;
270                     }
271                 }
272             }
273         }
274         return -1;
275     }
```

3. 测试结果（测试输入，测试输出）

(1) a 题

输入：

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


进程已结束，退出代码为 0
```

提交 OJ 的结果：

✓Accepted

#	Result	Score	Time	Memory
1	✓Accepted	10	0 ms	3496 KiB
2	✓Accepted	10	0 ms	3444 KiB
3	✓Accepted	10	0 ms	3404 KiB
4	✓Accepted	10	1 ms	3368 KiB
5	✓Accepted	10	1 ms	3428 KiB
6	✓Accepted	10	2 ms	3488 KiB
7	✓Accepted	10	2 ms	3564 KiB
8	✓Accepted	10	3 ms	3508 KiB
9	✓Accepted	10	4 ms	3564 KiB
10	✓Accepted	10	6 ms	3788 KiB

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 对于复杂的函数，我们可以将其分解成两个 protected 函数，然后通过 public 函数调用。这样能够减少耦合度。例子如下：

```
197     template<class T>
198     pair<int, vector<int>> Graph<T>::dfs(int s)
199     {
200         bool* jud = new bool[sizeofNode + 1]();//初始化
201         jud[s] = true;
202         int length = dfs_length(s, jud);//计算Length
203         for (int i = 0; i <= sizeofNode; i++)
204         {
205             jud[i] = false;//标记为false
206         }
207         vector<int>ans;
208         ans.push_back(s);
209         jud[s] = true;
210         dfs_sequence(s, &ans, jud);//保存序列
211         return pair<int, vector<int>>(length, ans);
212     }
```

(2) 要看好数据范围，比如下标是从 0 开始还是 1 开始很重要，如果我们数组开小了可能会导致我们 RE。
(3) 要保持好数据的一致性，比如本次实验中初始化的时候，外面给的是 n+1，但是里面在计算 size 的时候，也自增了 ，这就导致这个数据结构的 size 不对。最后会导致数组下标访问越界。
(4) 在提交 oj 的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。
(5) 对于 BOOL 数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为 bool 数组没有初始化导致 debug 了很久。

```
bool* jud_push = new bool[_size + 1]();
```

```
int* height = new int[_size + 1];
```

(6) 对于一个只有两个私有成员的 struct，我们可以直接使用 pair 来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了 Go 语言的一个原因。
(7) 对于 height 等操作的计算，一定要特判是否合法。因为对于 root 结点来说，它的父节点是自身，不能直接增加。

```
if(node.first.element!=node.second.element)//非根结点
{
    height[node.first.element] = max(height[node.first.element], height[node.secor
}
s.pop();
```

(8) 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：

```
40      while (currentNode != NULL &&//值不等于输入，且不越界，对于NULL值的判断要放到前边
41             currentNode->element.first != theKey)
42         currentNode = currentNode->next;
```

(9) 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：

```
112     if (p != NULL && p->element.first == theKey)
113     {
114         if (tp == NULL) firstNode = p->next;//头结点特殊处理
115         else tp->next = p->next;
116         delete p;//删除
117         dSize--;
118     }
119 }
```

(10)要注意私有成员的更新，public 函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的 Bug，最终 debug 查出来，就是下面这个：

```
    // switch to newQueue and set theFront and theBack
    theFront = 2 * arrayLength - 1;//更新私有成员
    theBack = arrayLength - 2;    // queue size arrayLength - 1
    arrayLength *= 2;
    queue = newQueue;//指针赋值
}

// put theElement at the theBack of the queue
theBack = (theBack + 1) % arrayLength;
queue[theBack] = theElement;
```

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>
#include <queue>
using namespace std;


template<class T>
class Graph
{
    chain<T>* ptr;//指针数组
    int sizeofNode;//结点数量
public:
    Graph(int n)
    {
        ptr = new chain<T>[n + 1];//从 1 开始
        sizeofNode = n;
    }
    virtual ~Graph();
    void insert(int i, int j);//插入
    void erase(int i, int j);//删除
    pair<int, vector<int>> Connected_Component()const;//联通分量，返回个数和联通集
    pair<int, vector<int>>dfs(int s);//返回 dfs 序列长度和字典序最小的 dfs 序列
    pair<int, vector<int>>bfs(int s)const;//返回 bfs 序列和字典序最小的 dfs 序列
    int distance(int i, int j)const;//返回 i 和 j 之间的长度
protected:
    void dfs_sequence(int s, vector<int>& v, bool* jud)const;
    int dfs_length(int s, bool* jud) const;

};
```

```
29.
30. template<class T>
31. void Graph<T>::insert(int i, int j)
32. {
33.     ptr[i].insert(j);
34.     ptr[j].insert(i);
35. }
36.
37. template<class T>
38. void Graph<T>::erase(int i, int j)
39. {
40.     ptr[i].erase(j);
41.     ptr[j].erase(i);
42. }
43.
44. template<class T>
45. pair<int, vector<int>> Graph<T>::Connected_Component() const
46. {
47.     int cnt = 0;
48.     queue<int>q;
49.     vector<int>sequence;
50.     bool* jud = new bool[1 + sizeofNode]();//初始化
51.     for (int i = 1; i <= sizeofNode; i++)
52.     {
53.         if (!jud[i])//没有被访问过
54.         {
55.             sequence.push_back(i);
56.             cnt++;
57.             q.push(i);
58.             jud[i] = true;
59.             while (!q.empty())
60.             {
61.                 int front = q.front();
62.                 q.pop();
63.                 //将临接的所有顶点压入
64.                 for (chainNode<T>* p = ptr[front].getFirstNode(); p; p = p->next)
65.                 {
66.                     if (!jud[p->element])
67.                     {
68.                         q.push(p->element);
69.                         jud[p->element] = true;
70.                     }
71.                 }
72.             }
73.         }
74.
```

```
    }
    return pair<int, vector<int>>(cnt, sequence);
}

template<class T>
int Graph<T>::dfs_length(int s, bool* jud) const
{
    int length = 1;
    for (chainNode<T>* p = ptr[s].getFirstNode(); p; p = p->next)
    {
        if (!jud[p->element])
        {
            jud[p->element] = true;
            length += dfs_length(p->element, jud);//递归调用
        }
    }
    return length;
}

template<class T>
void Graph<T>::dfs_sequence(int s, vector<int>& v, bool* jud) const
{
    for (chainNode<T>* p = ptr[s].getFirstNode(); p; p = p->next)
    {
        if (!jud[p->element])
            {
                jud[p->element] = true;
                v.push_back(p->element);
                dfs_sequence(p->element, v, jud);//递归调用
            }
        }
    }

    template<class T>
    pair<int, vector<int>> Graph<T>::dfs(int s)
    {
        bool* jud = new bool[sizeofNode + 1]();//初始化
        jud[s] = true;
        int length = dfs_length(s, jud);
        for (int i = 0; i <= sizeofNode; i++)
        {
            jud[i] = false;//标记为false
        }
        vector<int>ans;
        ans.push_back(s);
        jud[s] = true;
```

```
121.        dfs_sequence(s, ans, jud);
122.        return pair<int, vector<int>>(length, ans);
123.    }
124.
125.
126.
127.    template<class T>
128.    pair<int, vector<int>> Graph<T>::bfs(int s) const
129.    {
130.        int length = 1;
131.        bool* jud = new bool[sizeofNode + 1]();
132.        vector<int>sequence;//保存序列
133.        sequence.push_back(s);
134.        queue<int>q;
135.        q.push(s);
136.        jud[s] = true;
137.        while (!q.empty())
138.        {
139.            int front = q.front();
140.            q.pop();
141.            //压入临接顶点
142.            for (chainNode<T>* p = ptr[front].getFirstNode(); p; p = p->next)
143.            {
144.                if (!jud[p->element])
145.                {
146.                    jud[p->element] = true;
147.                    length += 1;//更新
148.                    sequence.push_back(p->element);
149.                    q.push(p->element);
150.                }
151.            }
152.        }
153.        return { length, sequence };//返回
154.    }
155.
156.    template<class T>
157.    int Graph<T>::distance(int i, int j) const//返回 i 和 j 之间的距离
158.    {
159.        int length = 0;
160.        bool* jud = new bool[sizeofNode + 1]();
161.        queue<int>q;
162.        q.push(i);//压入
163.        jud[i] = true;
164.        while (!q.empty())
165.        {
166.            int front = q.front();
```

```
167.            q.pop();
168.            if (front == j)
169.            {
170.                return length;
171.            }
172.            else
173.            {
174.                length++;
175.                for (chainNode<T>* p = ptr[front].getFirstNode(); p; p = p->next)
176.                {
177.                    if (!jud[p->element])
178.                    {
179.                        q.push(p->element);
180.                        jud[p->element] = true;
181.                    }
182.                }
183.            }
184.        }
185.        return -1;
186.    }
187.
188.    template<class T>
189.    Graph<T>::~Graph() {
190.
191.    }
192.
193.
194.
195.    template <class T>
196.    struct chainNode
197.    {//数据成员
198.        T element;
199.        chainNode<T>* next;
200.        //方法（三种）
201.        chainNode() {}
202.        chainNode(const T& element) { this->element = element; }
203.        chainNode(const T& element, chainNode<T>* next) { this->element = element; this->ne
        xt = next; }
204.    };
205.
206.    template <class T>
207.    class chain
208.    {
209.    protected:
210.        chainNode<T>* firstNode;//指向链表中第一个节点的指针
211.        int listSize;//线性表的元素个数
```

```
212.  public:
213.      //构造函数、析构函数
214.      chain(int initialcapacity = 10);
215.      ~chain();
216.      void erase(int& theElement);
217.      void insert(const T& theElement);
218.      chainNode<T>* getFirstNode() { return firstNode; }
219.
220.  };
221.
222.
223.  template <class T>
224.  chain<T>::chain(int initialcapacity)
225.  {//构造函数
226.      firstNode = nullptr;
227.      listSize = 0;
228.  }
229.
230.  template <class T>
231.  chain<T>::~chain()
232.  {//链表的析构函数，重复删除链表中的首节点直到全部删除
233.      while (firstNode != nullptr)
234.      {//删除首节点
235.          chainNode<T>* nextNode = firstNode->next;
236.          delete firstNode;
237.          firstNode = nextNode;
238.      }
239.  }
240.
241.
242.  template <class T>
243.  void chain<T>::erase(int& theElement)
244.  {//删除元素，保证有序
245.      chainNode<T>* p = firstNode, * tp = nullptr;
246.      while (p != nullptr && p->element < theElement)
247.      { //查找删除位置
248.          tp = p;
249.          p = p->next;
250.      }
251.      if (p != nullptr && p->element == theElement)
252.      {
253.          if (tp == nullptr)
254.              firstNode = p->next;
255.          else
256.              tp->next = p->next;
257.          listSize--;
```

```
            delete p;
        }
    }


template <class T>
void chain<T>::insert(const T& theElement)
{//插入元素，保证有序
    chainNode<T>* p = firstNode, * tp = nullptr;//儿子
    while (p != nullptr && p->element < theElement)
    { //查找插入位置
        tp = p;
        p = p->next;
    }
    if (p != nullptr && p->element == theElement)
    { //元素已经存在，则直接返回
        return;
    }
    //插入
    chainNode<T>* newNode = new chainNode<T>(theElement, p);
    if (tp == nullptr)
        firstNode = newNode;
    else
        tp->next = newNode;
    listSize++;
}


int main()
{
    int n, m, s, t;
    cin >> n >> m >> s >> t;
    Graph<int>g(n);
    for (int i = 0; i < m; i++)
    {
        int flag, from, to;
        cin >> flag >> from >> to;
        if (flag)
        {
            g.erase(from, to);//删除
        }
        else
        {
            g.insert(from, to);//插入
        }
    }
```

```
304.        pair<int, vector<int>>component = g.Connected_Component();//联通分量
305.        cout << component.first << endl;//联通分量个数
306.        for (int i = 0; i < component.second.size(); i++)
307.        {
308.            cout << component.second[i] << " ";//所有连通子图中最小点的编号
309.        }
310.        cout << endl;
311.        //dfs
312.        pair<int, vector<int>>dfs = g.dfs(s);
313.        cout << dfs.first << endl;//dfs 序列长度
314.        for (int i = 0; i < dfs.second.size(); i++)
315.        {
316.            cout << dfs.second[i] << " ";//字典序最小的 dfs 序列
317.        }
318.        cout << endl;
319.        //bfs
320.        pair<int, vector<int>>bfs = g.bfs(t);//bfs 序列长度
321.        cout << bfs.first << endl;
322.        for (int i = 0; i < bfs.second.size(); i++)
323.        {
324.            cout << bfs.second[i] << " ";//字典序最小的 bfs 序列
325.        }
326.        cout << endl;
327.        cout << g.distance(s, t);
328.
329.        return 0;
330.    }
```