实现对网络最大流与最小费用流问题的 solution 类

* 实验项目类型：设计性
*此表由学生按顺序填写

课程名称 面向对象程序设计/JAVA 语言 成绩评定
实验项目名称 实现对网络最大流与最小费用流问题的 solution 类
指导老师 干晓聪
实验项目编号 1 实验项目类型 设计性 实验地点 数学系机房
学生姓名 郭彦培 学号 2022101149
学院 信息科学技术学院 系 数学系 专业 信息管理与信息系统
实验时间 2023 年 11 月 1 日上午 ~ 2023 年 11 月 1 日中午

一、 实验目的

熟悉 ArrayList、PriorityQueue、LinkedList 等 java.util 提供的接口与类库，并且利用它们解决实际问题。

二、 实验环境

计算机：PC X64

操作系统：Windows

编程语言：JAVA

IDE：Visual Studio Code

三、 程序原理

对于最小费用最大流问题，首先给出问题定义：

给定一个网格$G = (V, E)$，每条边除了有容量限制$c(u, v)$，还有单位流量费用$w(u, v)$。

当(u, v)的流量为$f(u, v)$时，需要花费$w(u, v) \times f(u, v)$的费用。

w满足斜对称性，即$w(u,v) = -w(u,v)$。

该网络图中总花费最小的最大流则为最小费用最大流，即在最大化$\sum_{(s,u)\in E} f(s,u)$的前提下最小化$\sum_{(s,u)\in E} f(s,u) \times w(u,v)$。

此问题我们可以用 PD 算法（即$Peimal-Dual$原始对偶算法）在$O(mn + mlogmf)$内解决。

PD 的总体思路为首先利用 SPFA 算法（队列优化的$Bellman-Ford$最短路算法）为每个节点设置一个$potential\ h_u$ 用于将所有节点的权值转为非负数使得后续增广路计算时可以运用$Dijkstra$ 算法解决。

由于$Johnson$ 算法的正确性，我们可以类比地证明设置了势能的新网络上最短路与原网络一一对应。

每次求得增广边后增广图形态会产生变化，因此需要更新各个节点的势能。

设增广后源点到第i个节点的最短距离记为d_i，只需给对应的$potential\ h_i$加上d_i即可。

由于对于任意次增广后的残量网络，有$d_i' + \left(w(i,j) + h_i - h_j\right) = d_j'$,即新增的边权也是非负的。

接下来的思路与SSP算法相同，即每次寻找单位费用最小的增广路进行增广，直到图上不存在增广路为止。

由于我们已经对网络图的权值增加了$potential\ h_u$，因此可以用$Dijkstra$ 算法求解每一次的最小增广路。

四、 程序代码

文件`sis8\maxFlow.java`实现了一个`maxFlow`类

```
package sis8;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;


public class maxFlow {
    private class edge {
```

```
        public int nxt,                    // 出度
                   cap,                    // 容量
                   flow;                   // 流量
        public int[] revNodeIdx; // 反向边
        public edge(int _nxt, int _cap) {
            nxt = _nxt;
            cap = _cap;
            flow = 0;
        }
        public void setRevIdx(int _i, int _j) {
            revNodeIdx = new int[2];
            revNodeIdx[0] = _i;
            revNodeIdx[1] = _j;
        }
    }
    private ArrayList<ArrayList<edge>> edgeList; // 节点列表
    private ArrayList<Integer> dep;              // 深度
    private ArrayList<Integer> fir;              // 节点最近合法边
    private int maxFlowAns;

    private int T, S;

    public maxFlow(int _n) {
        maxFlowAns = 0;
        S = 1;
        T = _n;
        edgeList = new ArrayList<ArrayList<edge>>();
        for(int i = 0;i <= _n;i ++) edgeList.add(new ArrayList<edge>());
        dep = new ArrayList<Integer>();
        fir = new ArrayList<Integer>();
        for(int i = 0;i <= _n;i ++) dep.add(0);
        for(int i = 0;i <= _n;i ++) fir.add(0);
    }

    public void resetTS(int _T, int _S) {
        T = _T;
        S = _S;
    }

    public void addedge(int _u, int _v, int _w) {
        edgeList.get(_u).add(new edge(_v, _w));
        edgeList.get(_v).add(new edge(_u, 0)); // 反向建边
        edgeList.get(_u).get(edgeList.get(_u).size() - 1).setRevIdx(_v,
edgeList.get(_v).size() - 1);
```

```
        edgeList.get(_v).get(edgeList.get(_v).size() - 1).setRevIdx(_u,
edgeList.get(_u).size() - 1);
    }

    public boolean bfs() { // 统计深度
        Queue<Integer> que = new LinkedList<Integer>();
        for (int i = 0; i < dep.size(); i++)
            dep.set(i, 0);

        dep.set(S, 1);
        que.add(S);
        while (que.size() != 0) {
            int at = que.peek();
            que.poll();
            for (int i = 0; i < edgeList.get(at).size(); i++) {
                edge tar = edgeList.get(at).get(i);
                if ((dep.get(tar.nxt) == 0) && (tar.flow < tar.cap)) {
                    dep.set(tar.nxt, dep.get(at) + 1);
                    que.add(tar.nxt);
                }
            }
        }
        return dep.get(T) != 0;
    }

    public int dfs(int at, int flow) {
        if ((at == T) || (flow == 0))
            return flow; // 到了或者没了
        int ret = 0;  // 本节点最大流
        for (int i = fir.get(at); i < edgeList.get(at).size(); i++) {
            edge tar = edgeList.get(at).get(i);       // 目前遍历的边
            int tlFlow = 0;                           // 目前边的最大流
            if (dep.get(at) == dep.get(tar.nxt) - 1) { // 遍历到的边为合法边
                tlFlow = dfs(tar.nxt, Math.min(tar.cap - tar.flow, flow -
ret));
                if (tlFlow == 0)

continue;                                                          // 若最大
流为 0，什么都不做
                ret +=
tlFlow;                                                          // 本节点最大
流累加
                edgeList.get(at).get(i).flow +=
tlFlow;                                          // 本节点实时流量
```

```
                edgeList.get(tar.revNodeIdx[0]).get(tar.revNodeIdx[1]).flow
-= tlFlow; // 反向边流量
                if (ret == flow)
                    return ret; // 充满了就不继续扫了
            }
        }
        return ret;
    }

    public int dinic() {
        if (maxFlowAns != 0)
            return maxFlowAns;
        while (bfs()) {
            for(int i = 0;i < fir.size();i ++) fir.set(i, 0);
            maxFlowAns += dfs(S, Integer.MAX_VALUE);
        }
        return maxFlowAns;
    }

}
```

文件`sis8\minCost.java`实现了一个`minCost`类

```
package sis8;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class minCost { //JAVA version
    public static class edge {
        public int v, f, c, next;

        edge(int _v, int _f, int _c, int _next) {
            v = _v;
            f = _f;
            c = _c;
            next = _next;
        }

        edge() {
        }
```

```
    }

    private static final int INF = 0x3f3f3f3f;

    public static void vecset(int value, ArrayList<Integer> arr) {
        for (int i = 0; i < arr.size(); i++)
            arr.set(i, value);
        return;
    }

    public static class node {
        public int v, e;
    }

    public static class mypair implements Comparable<mypair> {
        public int dis, id;

        public mypair(int d, int x) {
            dis = d;
            id = x;
        }

        public int compareTo(mypair a) {
            return dis - a.dis;
        }
    }

    public ArrayList<Integer> head;
    public ArrayList<Integer> dis;
    public ArrayList<Integer> vis;
    public ArrayList<Integer> h;
    public ArrayList<edge> e;
    public ArrayList<node> p;
    public int n, m, s, t, cnt = 1, maxf, minc;

    public minCost(int _n, int _m, int _s, int _t) {
        n = _n;
        m = _m;
        s = _s;
        t = _t;
        maxf = 0;
        minc = 0;
        head = new ArrayList<Integer>();
        dis = new ArrayList<Integer>();
```

```
        vis = new ArrayList<Integer>();
        e = new ArrayList<edge>();
        h = new ArrayList<Integer>();
        p = new ArrayList<node>();
        for (int i = 0; i <= n + 2; i++) {
            head.add(0);
            dis.add(0);
            vis.add(0);
            h.add(0);
        }
        for (int i = 0; i <= m + 2; i++)
            p.add(new node());
    }

    public void addedge(int u, int v, int f, int c) {
        e.add(new edge(v, f, c, head.get(u)));
        head.set(u, e.size() - 1);
        e.add(new edge(u, 0, -c, head.get(v)));
        head.set(v, e.size() - 1);
    }

    public boolean dijkstra() {
        PriorityQueue<mypair> q = new PriorityQueue<mypair>();
        vecset(INF, dis);
        vecset(0, vis);
        dis.set(s, 0);
        q.add(new mypair(0, s));
        while (!q.isEmpty()) {
            int u = q.peek().id;
            q.poll();
            if (vis.get(u) == 1)
                continue;
            vis.set(u, 1);
            for (int i = head.get(u); i != 0; i = e.get(i).next) {
                int v = e.get(i).v, nc = e.get(i).c + h.get(u) - h.get(v);
                if (e.get(i).f != 0 && dis.get(v) > dis.get(u) + nc) {
                    dis.set(v, dis.get(u) + nc);
                    p.get(v).v = u;
                    p.get(v).e = i;
                    if (vis.get(v) != 1)
                        q.add(new mypair(dis.get(v), v));
                }
            }
        }
```

```
        return dis.get(t) != INF;
    }

    public void spfa() {
        Queue<Integer> q = new LinkedList<Integer>();
        vecset(63, h);
        h.set(s, 0);
        vis.set(s, 1);
        q.add(s);
        while (!q.isEmpty()) {
            int u = q.peek();
            q.poll();
            vis.set(u, 0);
            for (int i = head.get(u); i != 0; i = e.get(i).next) {
                int v = e.get(i).v;
                if (e.get(i).f != 0 && h.get(v) > h.get(u) + e.get(i).c) {
                    h.set(v, h.get(u) + e.get(i).c);
                    if (vis.get(v) != 1) {
                        vis.set(v, 1);
                        q.add(v);
                    }
                }
            }
        }
    }

    public int pd() {
        spfa();
        while (dijkstra()) {
            int minf = INF;
            for (int i = 1; i <= n; i++)
                h.set(i, h.get(i) + dis.get(i));
            for (int i = t; i != s; i = p.get(i).v)
                minf = Math.min(minf, e.get(p.get(i).e).f);
            for (int i = t; i != s; i = p.get(i).v) {
                e.get(p.get(i).e).f -= minf;
                e.get(p.get(i).e ^ 1).f += minf;
            }
            maxf += minf;
            minc += minf * h.get(t);
        }
        return 0;
    }
```

```
    public void printAns() {
        System.out.println(maxf + " " + minc);
    }

}
```

文件`sis8\solution.java`实现了一个`solution`类

```
package sis8;

public class solution {
    //构建带权图，求解最大流与最小费用流
    public static void main(String[] args) {
        try (java.util.Scanner sc = new java.util.Scanner(System.in)) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            int s = sc.nextInt();
            int t = sc.nextInt();
            minCost pd = new minCost(n, m, s, t);
            maxFlow mf = new maxFlow(n);
            for (int i = 1; i <= m; i++) {
                int u = sc.nextInt();
                int v = sc.nextInt();
                int c = sc.nextInt();
                int f = sc.nextInt();
                mf.addedge(u, v, c);
                pd.addedge(u, v, c, f);
            }
            pd.spfa();
            System.out.println(" 考虑费用的最小费用流 " + pd.maxf + " " +
pd.minc);
            mf.resetTS(t, s);
            mf.dinic();
            System.out.println(" 不考虑费用的最大流 " + mf.dinic());
        }
    }

}
```

五、 出现的问题、原因与解决方法

在实现最大流算法的过程中，发现`java.util`库中的`Queue`使用接口实现，无法直接实例化，于是查询资料得`Queue`需要借助`LinkedList`进行实例化。

经过一些思考并结合在后续实现过程中使用的感受，随后查询资料，我总结了一些`Queue`规定为接口的好处：

1. 抽象层次结构： 使用接口可以创建一个抽象的层次结构，使得可以定义多个具有不同实现的队列类型。这样，程序员可以根据具体的需求选择合适的实现，而不关心底层的数据结构或算法。
2. 多态性： 接口允许多个类实现相同的接口，这为多态性提供了支持。这意味着可以将不同实现的队列对象交替使用，而不需要修改调用这些队列的代码。
3. 解耦合： 使用接口可以将队列的定义与其具体的实现分离开来。这种解耦合使得可以更容易地更改或替换底层实现，而不会对使用队列的代码造成影响。
4. 一致的 API： 接口定义了队列应该具有的方法，这为实现提供了一致的 API。这使得可以编写通用的代码，而不用担心不同队列实现之间的差异。
5. 可扩展性： 如果需要在将来添加新的队列实现，只需实现 Queue 接口，而不必修改已有的代码。

六、 测试数据与运行结果

测试输入规则:

输入一行包含四个整数n, m, s, t，分别代表网络点数、边数、源点、汇点。接下来 m 行，每行四个整数u_i, v_i, w_i, c_i，分别代表第 i 条边的起点、终点、最大流量、单位费用。

输入	输出	解释
4 5 4 3 4 2 30 2 4 3 20 3 2 3 20 1 2 1 30 9 1 3 40 5	50 50 280	网络最大流为 50 考虑费用后最大流仍为 50，最小费用为 280