一、有向图

在实际生活中，很多应用相关的图都是有方向性的，最直观的就是网络，可以从A页面通过链接跳转到B页面，那么a和b连接的方向是a->b,但不能说是b->a,此时我们就需要使用有向图来解决这一类问题，它和我们之前学习的无向图，最大的区别就在于连接是具有方向的，在代码的处理上也会有很大的不同。

1.1 有向图的定义及相关术语

定义：

有向图是一副具有方向性的图，是由一组顶点和一组有方向的边组成的，每条方向的边都连着一对有序的顶点。

出度：

由某个顶点指出的边的个数称为该顶点的出度。

入度：

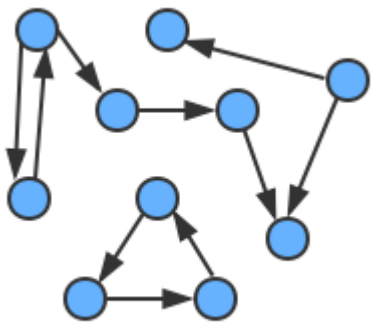
指向某个顶点的边的个数称为该顶点的入度。

有向路径：

由一系列顶点组成，对于其中的每个顶点都存在一条有向边，从它指向序列中的下一个顶点。

有向环：

一条至少含有一条边，且起点和终点相同的有向路径。

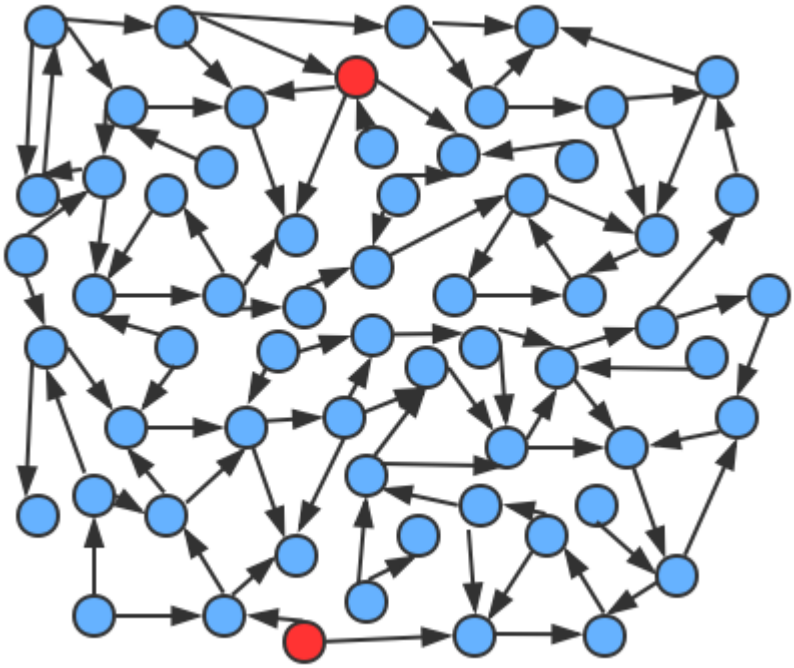


一副有向图中两个顶点v和w可能存在以下四种关系：

1. 没有边相连；

2. 存在从v到w的边v—>w;
3. 存在从w到v的边w—>v;
4. 既存在w到v的边，也存在v到w的边，即双向连接；

理解有向图是一件比较简单的，但如果要通过眼睛看出复杂有向图中的路径就不是那么容易了。



1.2 有向图API设计

类名	Digraph
构造方法	Digraph(int V)：创建一个包含V个顶点但不包含边的有向图
成员方法	1.public int V():获取图中顶点的数量 2.public int E():获取图中边的数量 3.public void addEdge(int v,int w):向有向图中添加一条边 v->w 4.public Queue adj(int v)：获取由v指出的边所连接的所有顶点 5.private Digraph reverse():该图的反向图
成员变量	1.private final int V: 记录顶点数量 2.private int E: 记录边数量 3.private Queue[] adj: 邻接表

在api中设计了一个反向图，其因为有向图的实现中，用adj方法获取出来的是由当前顶点v指向的其他顶点，如果能得到其反向图，就可以很容易得到指向v的其他顶点。

1.3 有向图实现

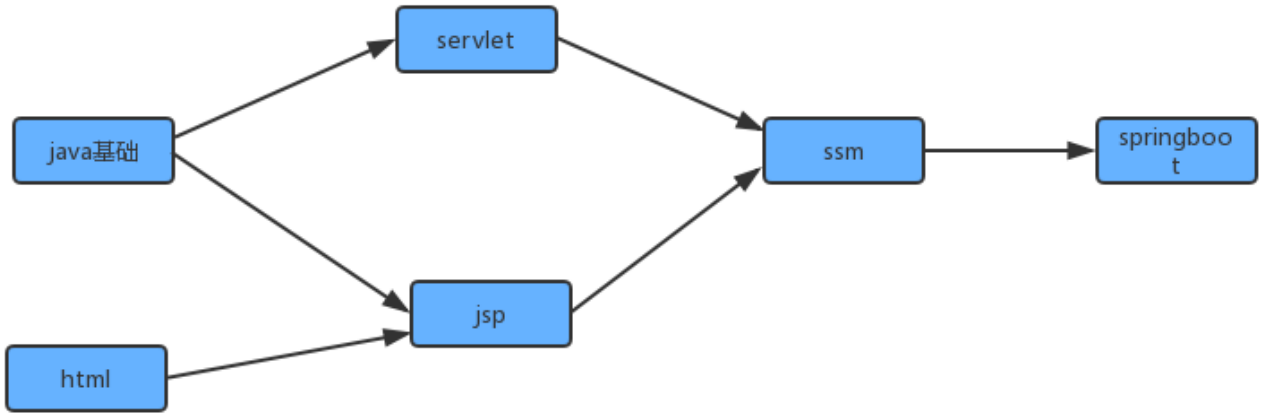
```
1  public class Digraph {
```

```java
    //顶点数目
    private final int V;
    //边的数目
    private int E;
    //邻接表
    private Queue<Integer>[] adj;

    public Digraph(int V){
        //初始化顶点数量
        this.V = V;
        //初始化边的数量
        this.E=0;
        //初始化邻接表
        this.adj = new Queue[V];
        //初始化邻接表中的空队列
        for (int i = 0; i < adj.length; i++) {
            adj[i] = new Queue<Integer>();
        }
    }


    //获取顶点数目
    public int V(){
        return V;
    }

    //获取边的数目
    public int E(){
        return E;
    }

    //向有向图中添加一条边 v->w
    public void addEdge(int v, int w) {
        //由于有向图中边是有向的，v->w 边，只需要让w出现在v的邻接表中，而不需要让v出现在w的邻接表中
        adj[v].enqueue(w);
        //边的数目自增1
        E++;
    }

    //获取由v指出的边所连接的所有顶点
    public Queue<Integer> adj(int v){
        return adj[v];
    }
    //该图的反向图
    private Digraph reverse(){
        //创建新的有向图对象
        Digraph r = new Digraph(V);
        //遍历0~V-1所有顶点,拿到每一个顶点v
        for (int v=0;v<V;v++){
            //得到原图中的v顶点对应的邻接表,原图中的边为 v->w,则反向图中边为w->v;
            for (Integer w : adj(v)) {
                r.addEdge(w,v);

            }
```

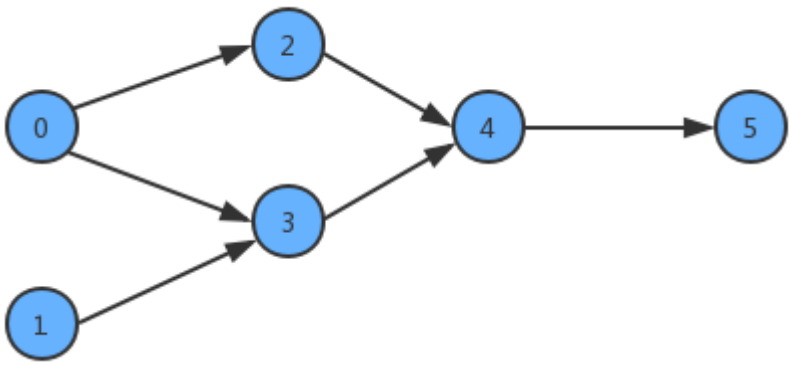
```
        }
        return r;
    }

}
```

二、拓扑排序

在现实生活中，我们经常会同一时间接到很多任务去完成，但是这些任务的完成是有先后次序的。以我们学习java学科为例，我们需要学习很多知识，但是这些知识在学习的过程中是需要按照先后次序来完成的。从java基础，到jsp/servlet，到ssm，到springboot等是个循序渐进且有依赖的过程。在学习jsp前要首先掌握java基础和html基础，学习ssm框架前要掌握jsp/servlet之类才行。



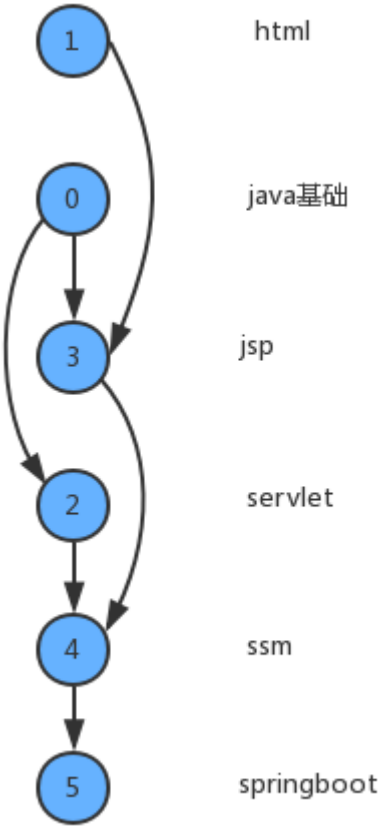
为了简化问题，我们使用整数为顶点编号的标准模型来表示这个案例：



此时如果某个同学要学习这些课程，就需要指定出一个学习的方案，我们只需要对图中的顶点进行排序，让它转换为一个线性序列，就可以解决问题，这时就需要用到一种叫**拓扑排序**的算法。

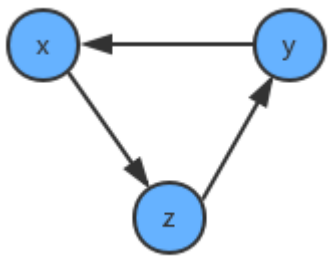
拓扑排序：

给定一副有向图，将所有的顶点排序，使得所有的有向边均从排在前面的元素指向排在后面的元素，此时就可以明确的表示出每个顶点的优先级。下列是一副拓扑排序后的示意图：



2.1 检测有向图中的环

如果学习x课程前必须先学习y课程，学习y课程前必须先学习z课程，学习z课程前必须先学习x课程，那么一定是有问题了，我们就没有办法学习了，因为这三个条件没有办法同时满足。其实这三门课程x、y、z的条件组成了一个环：



因此，如果我们要使用拓扑排序解决优先级问题，首先得保证图中没有环的存在。

2.1.1 检测有向环的API设计

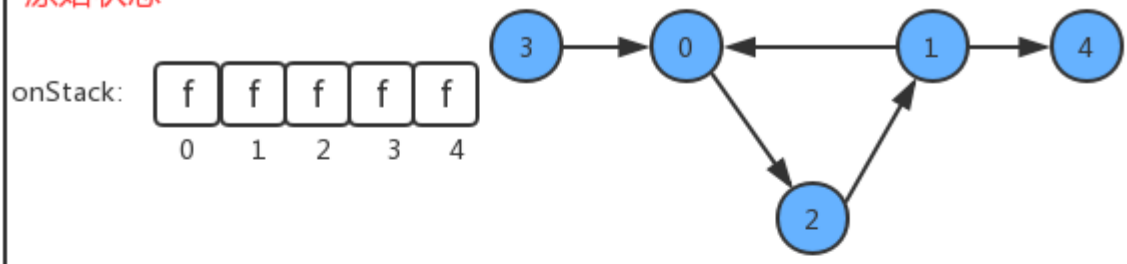
类名	DirectedCycle
构造方法	DirectedCycle(Digraph G)：创建一个检测环对象，检测图G中是否有环
成员方法	1.private void dfs(Digraph G,int v)：基于深度优先搜索，检测图G中是否有环 2.public boolean hasCycle():判断图中是否有环
成员变量	1.private boolean[] marked: 索引代表顶点，值表示当前顶点是否已经被搜索 2.private boolean hasCycle: 记录图中是否有环 3.private boolean[] onStack:索引代表顶点，使用栈的思想，记录当前顶点有没有已经处于正在搜索的有向路径上

2.1.2 检测有向环实现

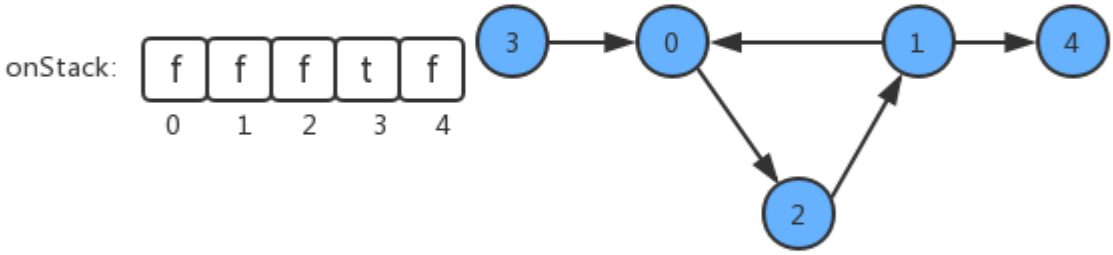
在API中添加了onStack[] 布尔数组，索引为图的顶点，当我们深度搜索时：

1. 在如果当前顶点正在搜索，则把对应的onStack数组中的值改为true，标识进栈；
2. 如果当前顶点搜索完毕，则把对应的onStack数组中的值改为false，标识出栈；
3. 如果即将要搜索某个顶点，但该顶点已经在栈中，则图中有环；

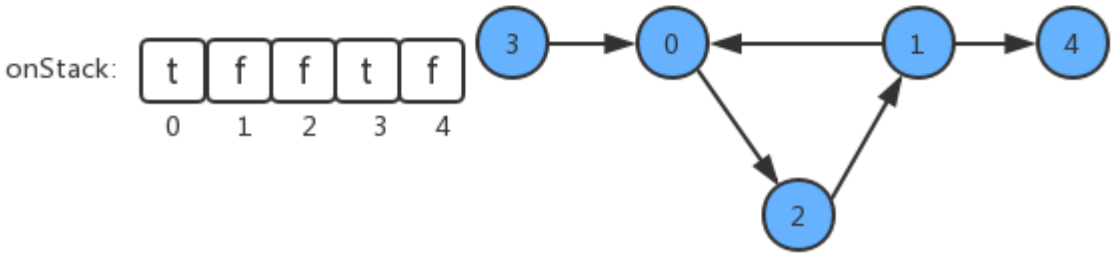
原始状态



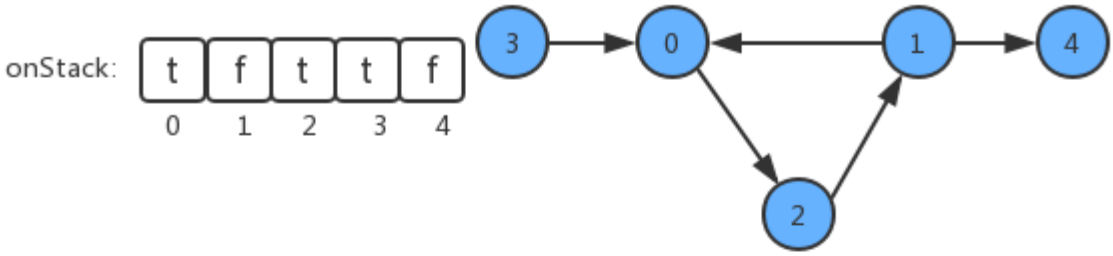
以顶点3作为入口，进行搜索，则3进栈



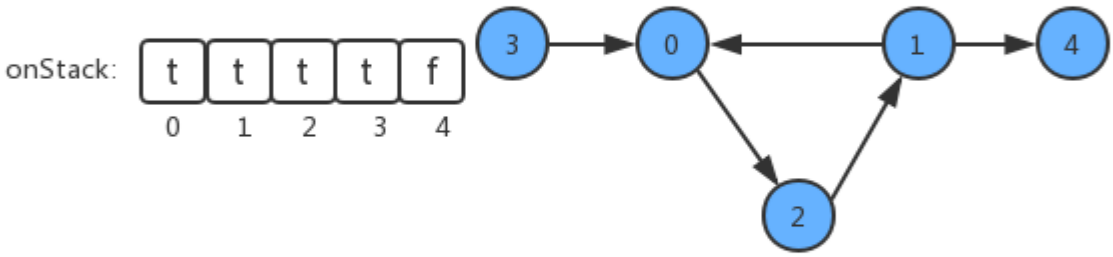
找到顶点3的邻接表，继续搜索0，0当前没有在栈中，则0进栈



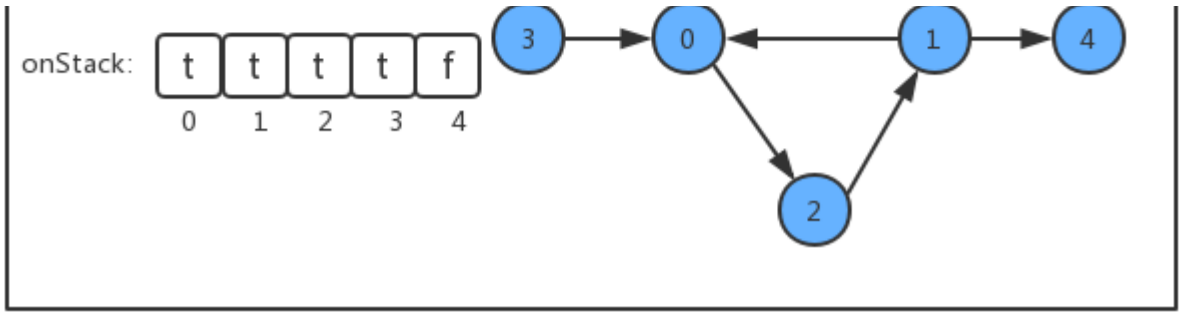
找到顶点0的邻接表，继续搜索2，2当前没有在栈中，则2进栈



找到顶点2的邻接表，继续搜索1，1当前没有在栈中，则1进栈



找到顶点1的邻接表，继续搜索0，0当前已经进栈，则图中有环，检测结束



代码：

```
public class DirectedCycle {
    //索引代表顶点，值表示当前顶点是否已经被搜索
    private boolean[] marked;
    //记录图中是否有环
    private boolean hasCycle;
    //索引代表顶点，使用栈的思想，记录当前顶点有没有已经处于正在搜索的有向路径上
    private boolean[] onStack;

    //创建一个检测环对象，检测图G中是否有环
    public DirectedCycle(Digraph G){
        //创建一个和图的顶点数一样大小的marked数组
        marked = new boolean[G.V()];
        //创建一个和图的顶点数一样大小的onStack数组
        onStack = new boolean[G.V()];
        //默认没有环
        this.hasCycle=false;
        //遍历搜索图中的每一个顶点
        for (int v = 0; v <G.V(); v++) {
            //如果当前顶点没有搜索过，则搜索
            if (!marked[v]){
                dfs(G,v);
            }
        }
    }

    //基于深度优先搜索，检测图G中是否有环
    private void dfs(Digraph G, int v){
        //把当前顶点标记为已搜索
        marked[v]=true;
        //让当前顶点进栈
        onStack[v]=true;
        //遍历v顶点的邻接表，得到每一个顶点w
        for (Integer w : G.adj(v)){
            //如果当前顶点w没有被搜索过，则递归搜索与w顶点相通的其他顶点
            if (!marked[w]){
                dfs(G,w);
            }

            //如果顶点w已经被搜索过，则查看顶点w是否在栈中，如果在，则证明图中有环，修改hasCycle标记，结束循环

            if (onStack[w]){
```

```
                hasCycle=true;
                return;
            }
        }
        //当前顶点已经搜索完毕，让当前顶点出栈
        onStack[v]=false;
    }

    //判断w顶点与s顶点是否相通
    public boolean hasCycle(){
        return hasCycle;
    }

}


//测试代码
public class DirectedCycleTest {
    public static void main(String[] args) throws Exception {
        //创建输入流
        BufferedReader reader = new BufferedReader(new
InputStreamReader(DirectedCycleTest.class.getClassLoader().getResourceAsStream("cycle_test.t
xt")));
        //读取顶点个数，初始化Graph图
        int number = Integer.parseInt(reader.readLine());
        Digraph G = new Digraph(number);
        //读取边的个数
        int roadNumber = Integer.parseInt(reader.readLine());
        //读取边，并调用addEdge方法
        for (int i = 0; i < roadNumber; i++) {
            String line = reader.readLine();
            int p = Integer.parseInt(line.split(" ")[0]);
            int q = Integer.parseInt(line.split(" ")[1]);
            G.addEdge(p, q);
        }

        //创建测试检测环对象
        DirectedCycle cycle = new DirectedCycle(G);
        //输出图中是否有环
        System.out.println(cycle.hasCycle());
    }

}
```

2.2 基于深度优先的顶点排序

如果要把图中的顶点生成线性序列其实是一件非常简单的事，之前我们学习并使用了多次深度优先搜索，我们会发现其实深度优先搜索有一个特点，那就是在一个连通子图上，每个顶点只会被搜索一次，如果我们能在深度优先搜索的基础上，添加一行代码，只需要将搜索的顶点放入到线性序列的数据结构中，我们就能完成这件事。

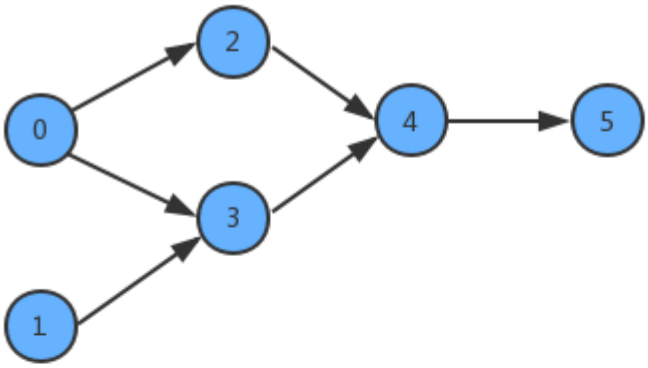
2.2.1 顶点排序API设计

类名	DepthFirstOrder
构造方法	DepthFirstOrder(Digraph G)：创建一个顶点排序对象，生成顶点线性序列；
成员方法	1.private void dfs(Digraph G,int v)：基于深度优先搜索，生成顶点线性序列 2.public Stack reversePost():获取顶点线性序列
成员变量	1.private boolean[] marked: 索引代表顶点，值表示当前顶点是否已经被搜索 2.private Stack reversePost: 使用栈，存储顶点序列

2.2.2 顶点排序实现

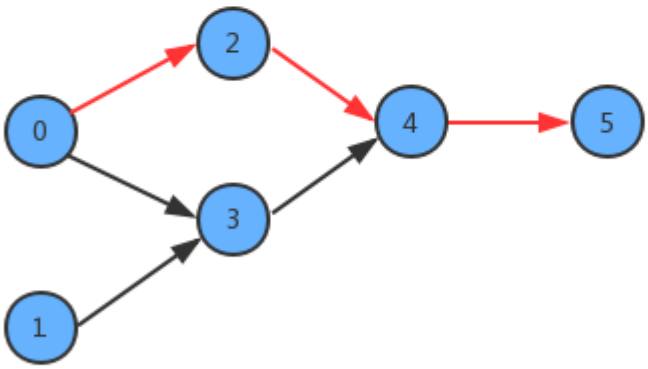
在API的设计中，我们添加了一个栈reversePost用来存储顶点，当我们深度搜索图时，每搜索完毕一个顶点，把该顶点放入到reversePost中，这样就可以实现顶点排序。

原始图

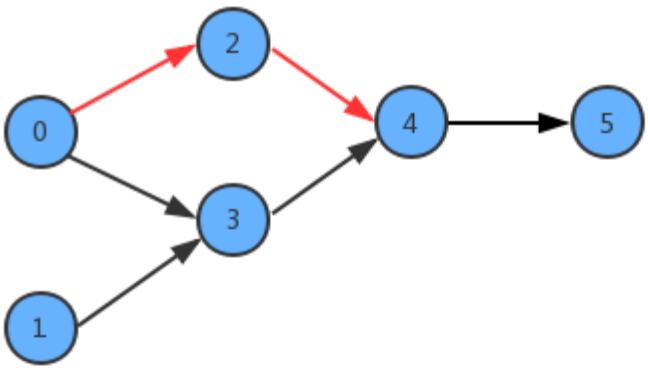


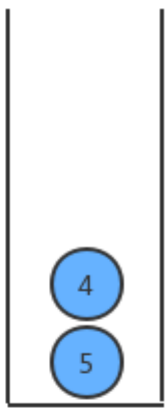
原始栈

以0顶点为入口，开始搜索，递归搜索一直找到顶点5,顶点5邻接表中没有数据，所以顶点5搜索完毕，入栈

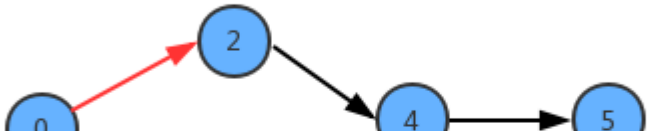


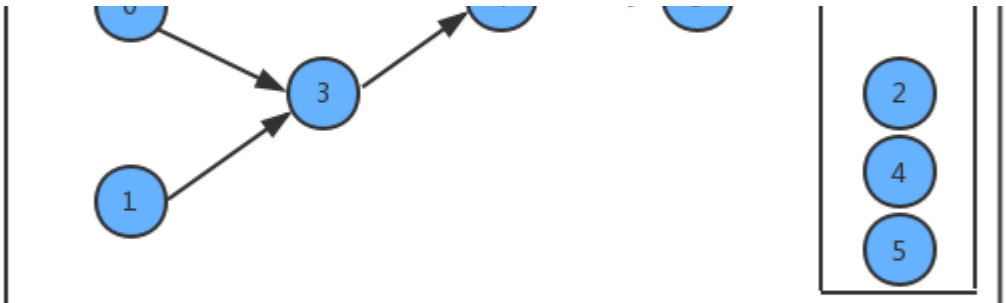
返回到顶点4，邻接表中没有其他顶点，4 搜索结束，顶点4入栈



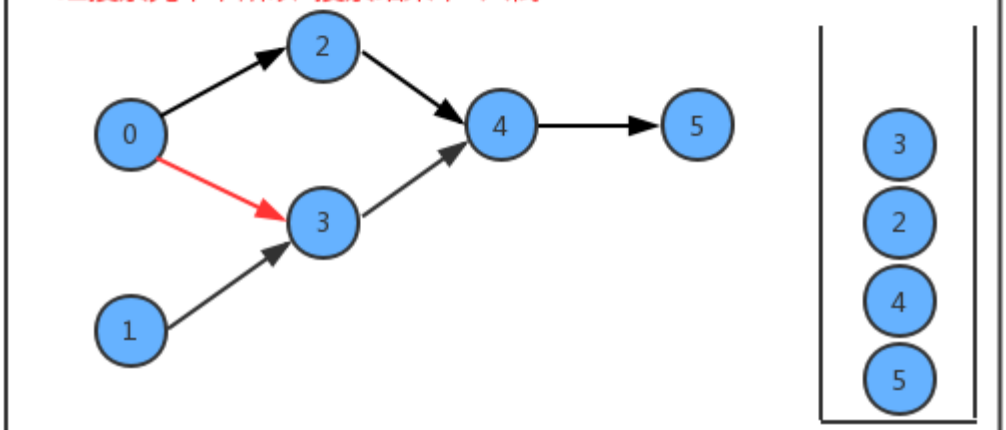


返回到顶点2，邻接表中没有其他顶点，2 搜索结束，顶点2入栈

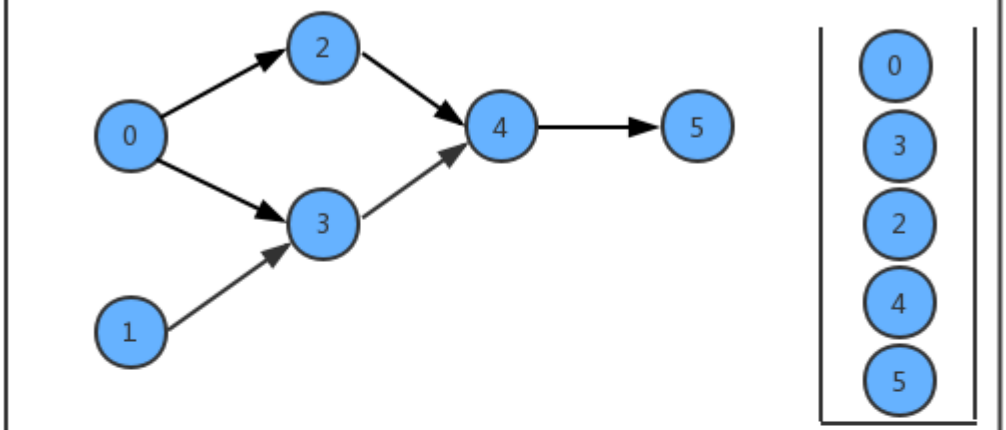




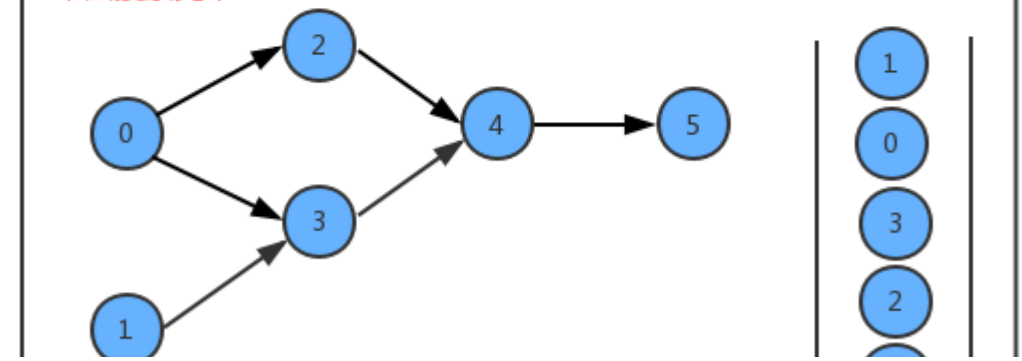
返回到顶点1，继续找邻接表中的顶点3搜索，3邻接表中的顶点4已经搜索完毕，所以3搜索结束，3入栈



返回到顶点0，邻接表中没有其他顶点，0搜索结束，0入栈



继续以1作为入口，临街表中的3已经搜索，1搜索完毕，1入栈，整个图搜索完毕



代码：

```
public class DepthFirstOrder {
    //索引代表顶点，值表示当前顶点是否已经被搜索
    private boolean[] marked;
    //使用栈，存储顶点序列
    private Stack<Integer> reversePost;

    //创建一个检测环对象，检测图G中是否有环
    public DepthFirstOrder(Digraph G){
        //创建一个和图的顶点数一样大小的marked数组
        marked = new boolean[G.V()];
        reversePost = new Stack<Integer>();
        //遍历搜索图中的每一个顶点
        for (int v = 0; v <G.V(); v++) {
            //如果当前顶点没有搜索过，则搜索
            if (!marked[v]){
                dfs(G,v);
            }
        }
    }

    //基于深度优先搜索，检测图G中是否有环
    private void dfs(Digraph G, int v){
        //把当前顶点标记为已搜索
        marked[v]=true;
        //遍历v顶点的邻接表，得到每一个顶点w
        for (Integer w : G.adj(v)){
            //如果当前顶点w没有被搜索过，则递归搜索与w顶点相通的其他顶点
            if (!marked[w]){
                dfs(G,w);
            }
        }
        //当前顶点已经搜索完毕，让当前顶点入栈
        reversePost.push(v);
    }

    //获取顶点线性序列
    public Stack<Integer>  reversePost(){
        return reversePost;
    }
}
```

2.3 拓扑排序实现

前面已经实现了环的检测以及顶点排序，那么拓扑排序就很简单了，基于一幅图，先检测有没有环，如果没有环，则调用顶点排序即可。

API设计：

类名	TopoLogical
构造方法	TopoLogical(Digraph G)：构造拓扑排序对象
成员方法	1.public boolean isCycle()：判断图G是否有环 2.public Stack order():获取拓扑排序的所有顶点
成员变量	1.private Stack order: 顶点的拓扑排序

代码：

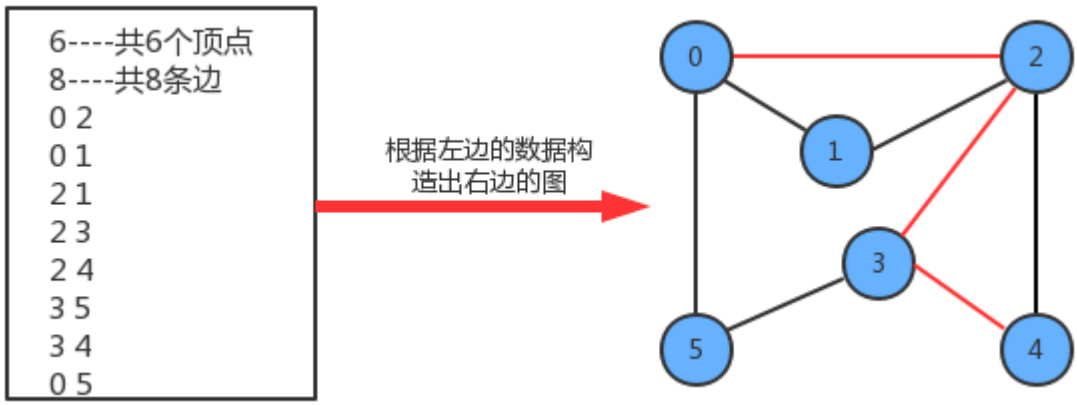
```
public class TopoLogical {
    //顶点的拓扑排序
    private Stack<Integer> order;

    //构造拓扑排序对象
    public TopoLogical(Digraph G) {
        //创建检测环对象，检测图G中是否有环
        DirectedCycle dCycle = new DirectedCycle(G);
        if (!dCycle.hasCycle()){
            //如果没有环，创建顶点排序对象，进行顶点排序
            DepthFirstOrder depthFirstOrder = new DepthFirstOrder(G);
            order = depthFirstOrder.reversePost();
        }
    }

    //判断图G是否有环
    private boolean isCycle(){
        return order==null;
    }

    //获取拓扑排序的所有顶点
    public Stack<Integer>  order(){
        return order;
    }
}


//测试代码
public class TopoLogicalTest {

    public static void main(String[] args) throws Exception {
        //创建输入流
        BufferedReader reader = new BufferedReader(new
InputStreamReader(TopoLogicalTest.class.getClassLoader().getResourceAsStream("topological_te
st.txt")));
        //读取顶点个数，初始化Graph图
```

```
        int number = Integer.parseInt(reader.readLine());
        Digraph G = new Digraph(number);
        //读取边的个数
        int roadNumber = Integer.parseInt(reader.readLine());
        //读取边，并调用addEdge方法
        for (int i = 0; i < roadNumber; i++) {
            String line = reader.readLine();
            int p = Integer.parseInt(line.split(" ")[0]);
            int q = Integer.parseInt(line.split(" ")[1]);
            G.addEdge(p, q);
        }

        //创建拓扑排序对象对象
        TopoLogical topo = new TopoLogical(G);
        Stack<Integer> order = topo.order();
        //遍历打印
        StringBuilder sb = new StringBuilder();
        for (Integer v : order) {
            sb.append(v+"->");

        }

        sb.deleteCharAt(sb.length()-1);
        sb.deleteCharAt(sb.length()-1);
        System.out.println(sb);
    }


}
```

三、加权无向图

加权无向图是一种为每条边关联一个权重值或是成本的图模型。这种图能够自然地表示许多应用。在一副航空图中，边表示航线，权值则可以表示距离或是费用。在一副电路图中，边表示导线，权值则可能表示导线的长度即成本，或是信号通过这条先所需的时间。此时我们很容易就能想到，最小成本的问题，例如，从西安飞纽约，怎样飞才能使时间成本最低或者是金钱成本最低？

在下图中，从顶点0到顶点4有三条路径，分别为0-2-3-4,0-2-4,0-5-3-4,那我们如果要通过那条路径到达4顶点最好呢？此时就要考虑，那条路径的成本最低。



3.1 加权无向图边的表示

加权无向图中的边我们就不能简单的使用v-w两个顶点表示了，而必须要给边关联一个权重值，因此我们可以使用对象来描述一条边。

API设计：

类名	Edge implements Comparable
构造方法	Edge(int v,int w,double weight)：通过顶点v和w，以及权重weight值构造一个边对象
成员方法	1.public double weight():获取边的权重值 2.public int either():获取边上的一个点 3.public int other(int vertex)):获取边上除了顶点vertex外的另外一个顶点 4.public int compareTo(Edge that)：比较当前边和参数that边的权重，如果当前边权重大，返回1，如果一样大，返回0，如果当前权重小，返回-1
成员变量	1.private final int v：顶点一 2.private final int w：顶点二 3.private final double weight：当前边的权重

代码：

```
public class Edge implements Comparable<Edge> {
    private final int v;//顶点一
    private final int w;//顶点二
    private final double weight;//当前边的权重

    //通过顶点v和w，以及权重weight值构造一个边对象
    public Edge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }


    //获取边的权重值
```

```
    public double weight(){
        return weight;
    }

    //获取边上的一个点
    public int either(){
      return v;
    }

    //获取边上除了顶点vertex外的另外一个顶点
    public int other(int vertex){

        if (vertex==v){
            //如果传入的顶点vertext是v，则返回另外一个顶点w
            return w;
        }else{
            //如果传入的顶点vertext不是v，则返回v即可
            return v;
        }
    }

    @Override
    public int compareTo(Edge that) {

        int cmp;
        if (this.weight()>that.weight()){
            //如果当前边的权重大于参数边that的权重，返回1
            cmp=1;
        }else if(this.weight()<that.weight()){
            //如果当前边的权重小于参数边that的权重，返回-1
            cmp=-1;
        }else{
            //如果当前边的权重等于参数边that的权重，返回0
            cmp=0;
        }
        return cmp;
    }
}
```

3.2 加权无向图的实现

之前我们已经完成了无向图，在无向图的基础上，我们只需要把边的表示切换成Edge对象即可。

API设计：

类名	EdgeWeightedGraph
构造方法	EdgeWeightedGraph(int V)：创建一个含有V个顶点的空加权无向图
成员方法	1.public int V():获取图中顶点的数量 2.public int E():获取图中边的数量 3.public void addEdge(Edge e):向加权无向图中添加一条边e 4.public Queue adj(int v)：获取和顶点v关联的所有边 5.public Queue edges()：获取加权无向图的所有边
成员变量	1.private final int V: 记录顶点数量 2.private int E: 记录边数量 3.private Queue[] adj: 邻接表

代码：

```
public class EdgeWeightedGraph {
    //顶点总数
    private final int V;
    //边的总数
    private int E;
    //邻接表
    private Queue<Edge>[] adj;

    //创建一个含有V个顶点的空加权无向图
    public EdgeWeightedGraph(int V) {
        //初始化顶点数量
        this.V = V;
        //初始化边的数量
        this.E = 0;
        //初始化邻接表
        this.adj = new Queue[V];
        //初始化邻接表中的空队列
        for (int i = 0; i < adj.length; i++) {
            adj[i] = new Queue<Edge>();
        }

    }

    //获取图中顶点的数量
    public int V() {
        return V;
    }

    //获取图中边的数量
    public int E() {
        return E;
    }


```

```
    //向加权无向图中添加一条边e
    public void addEdge(Edge e) {
        //获取边中的一个顶点v
        int v = e.either();
        //获取边中的另一个顶点w
        int w = e.other(v);
        //因为是无向图，所以边e需要同时出现在两个顶点的邻接表中
        adj[v].enqueue(e);
        adj[w].enqueue(e);

        //边的数量+1
        E++;
    }

    //获取和顶点v关联的所有边
    public Queue<Edge> adj(int v) {
        return adj[v];
    }

    //获取加权无向图的所有边
    public Queue<Edge> edges() {
        //创建一个队列，存储所有的边
        Queue<Edge> allEdge = new Queue<>();
        //遍历顶点，拿到每个顶点的邻接表
        for (int v = 0; v < this.V; v++) {
            //遍历邻接表，拿到邻接表中的每条边
            for (Edge e : adj(v)) {
                /*
                因为无向图中，每条边对象Edge都会在两个顶点的邻接表中各出现一次，为了不重复获取，暂定一条规则：
                    除了当前顶点v，再获取边e中的另外一个顶点w，如果v<w则添加，这样可以保证同一条边只会被统计一次
                 */
                if (e.other(v) < v) {
                    allEdge.enqueue(e);
                }
            }
        }
        return allEdge;
    }
}
```

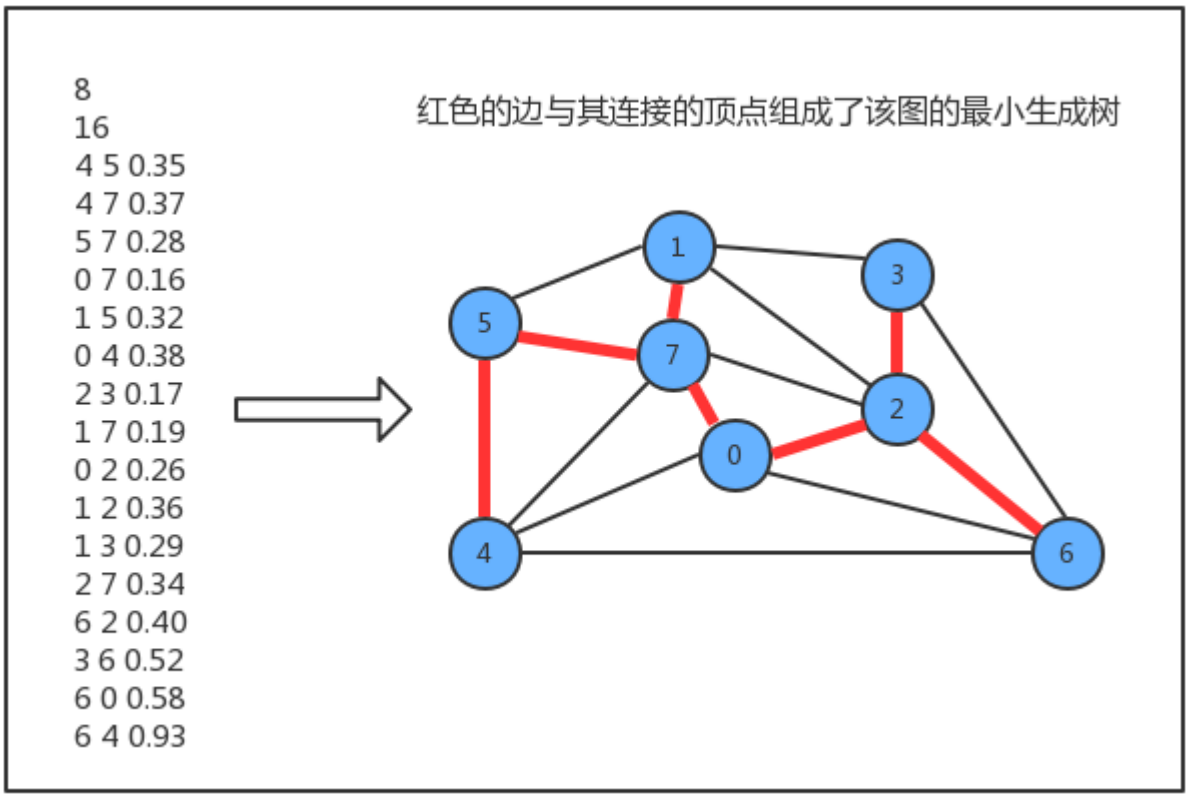
四、最小生成树

之前学习的加权图，我们发现它的边关联了一个权重，那么我们就可以根据这个权重解决最小成本问题，但如何才能找到最小成本对应的顶点和边呢？最小生成树相关算法可以解决。

4.1 最小生成树定义及相关约定

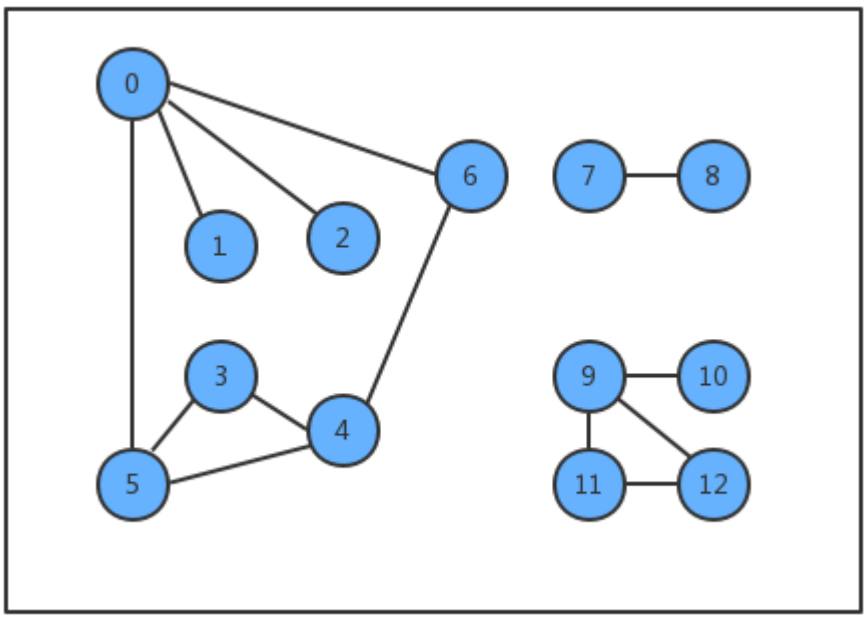
定义：

图的生成树是它的一棵含有其所有顶点的无环连通子图，一副加权无向图的最小生成树它的一棵权值(树中所有边的权重之和)最小的生成树



约定：

只考虑连通图。最小生成树的定义说明它只能存在于连通图中，如果图不是连通的，那么分别计算每个连通图子图的最小生成树，合并到一起称为最小生成森林。

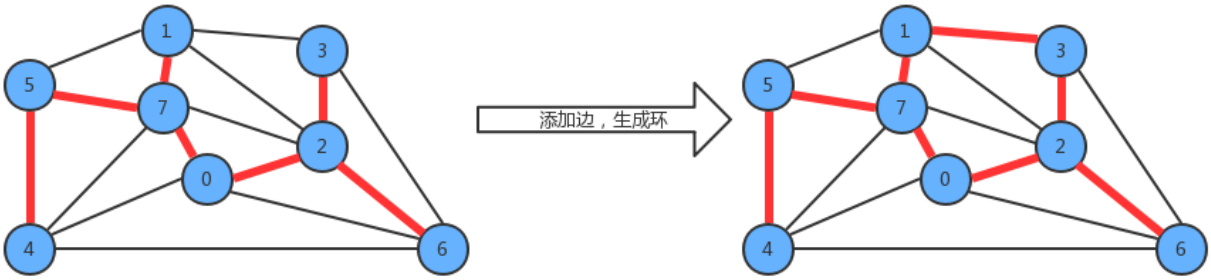


所有边的权重都各不相同。如果不同的边权重可以相同，那么一副图的最小生成树就可能不唯一了，虽然我们的算法可以处理这种情况，但为了好理解，我们约定所有边的权重都各不相同。

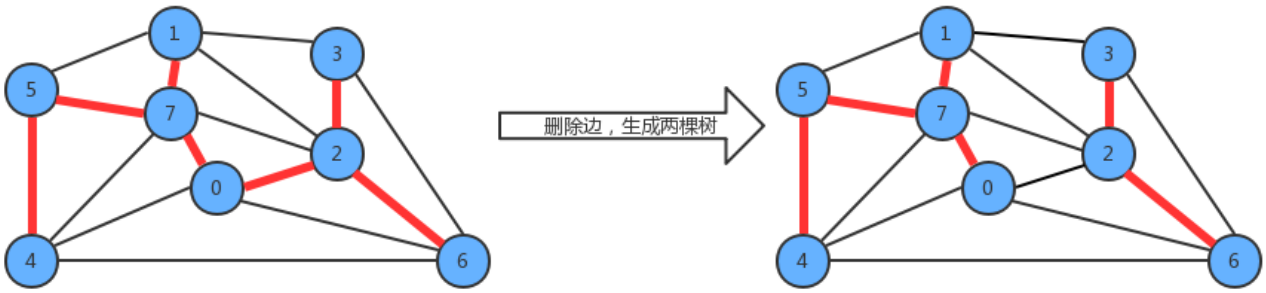
4.2 最小生成树原理

4.2.1 树的性质

1. 用一条边接树中的任意两个顶点都会产生一个新的环；



2. 从树中删除任意一条边，将会得到两棵独立的树；



4.2.2 切分定理

要从一副连通图中找出该图的最小生成树，需要通过切分定理完成。

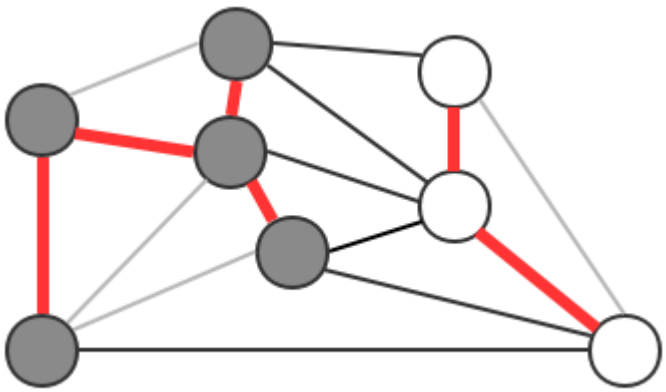
切分：

将图的所有顶点按照某些规则分为两个非空且没有交集的集合。

横切边：

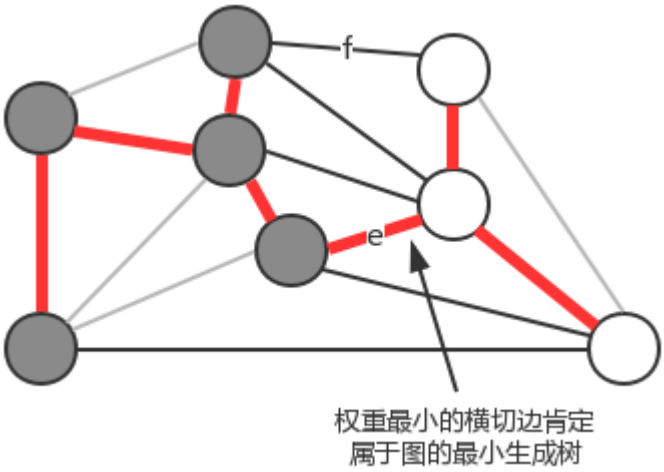
连接两个属于不同集合的顶点的边称之为横切边。

例如我们将图中的顶点切分为两个集合，灰色顶点属于一个集合，白色顶点属于另外一个集合，那么效果如下：

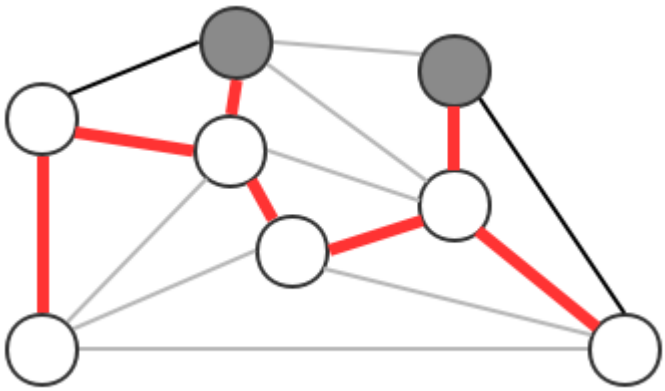


切分定理：

在一副加权图中，给定任意的切分，它的横切边中的权重最小者必然属于图中的最小生成树。

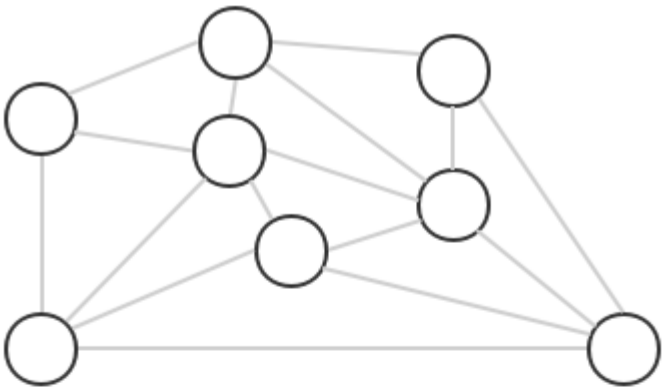


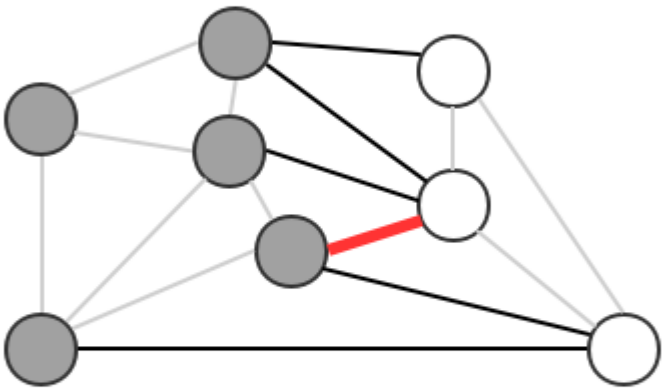
注意:一次切分产生的多个横切边中，权重最小的边不一定是所有横切边中唯一属于图的最小生成树的边。

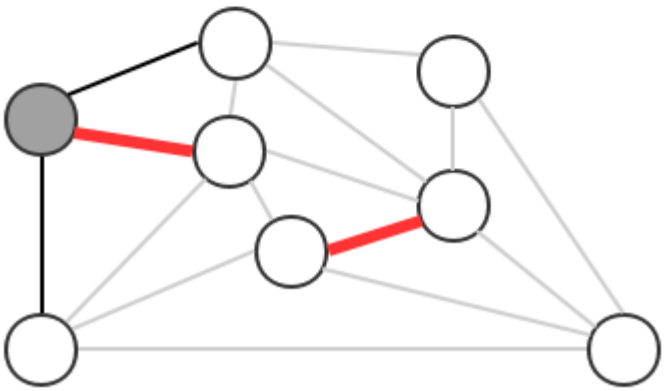


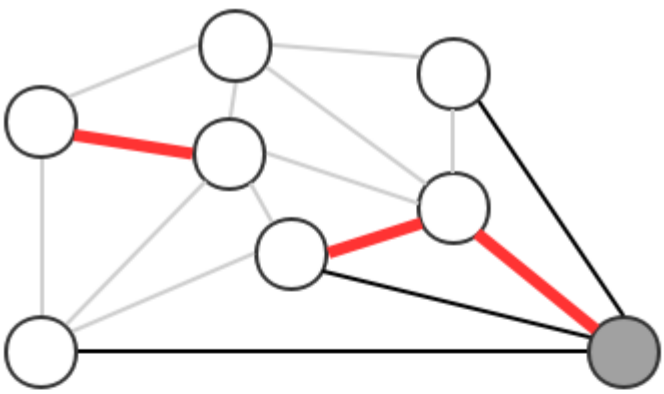
4.3 贪心算法

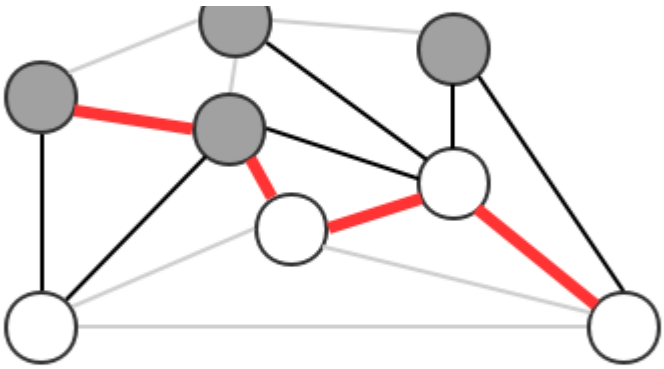
贪心算法是计算图的最小生成树的基础算法，它的基本原理就是切分定理，**使用切分定理找到最小生成树的一条边，不断的重复直到找到最小生成树的所有边。**如果图有V个顶点，那么需要找到V-1条边，就可以表示该图的最小生成树。

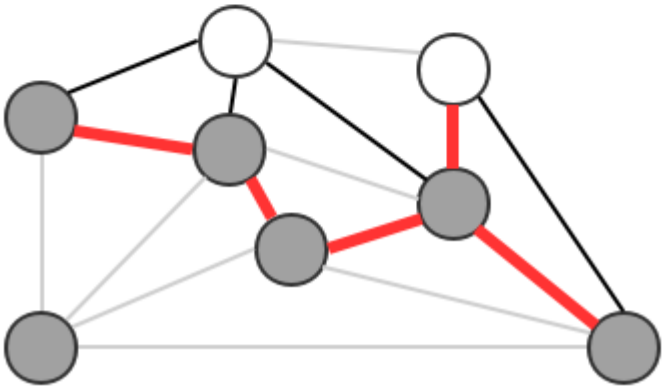


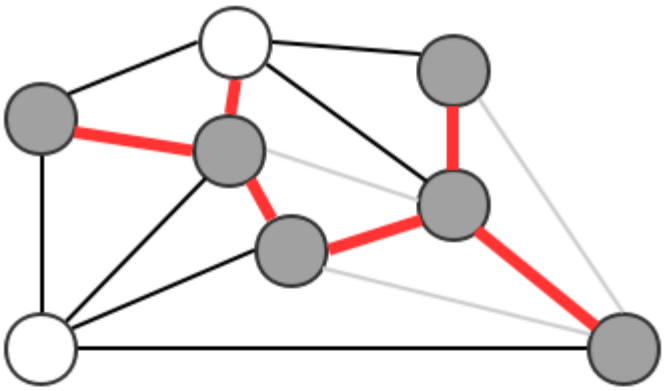


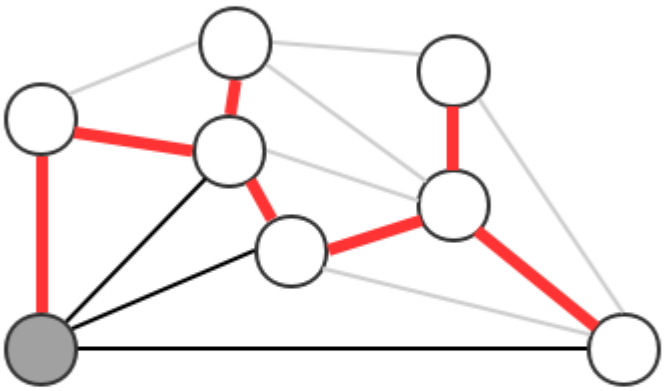












最小生成树找到了，包含V-1条边

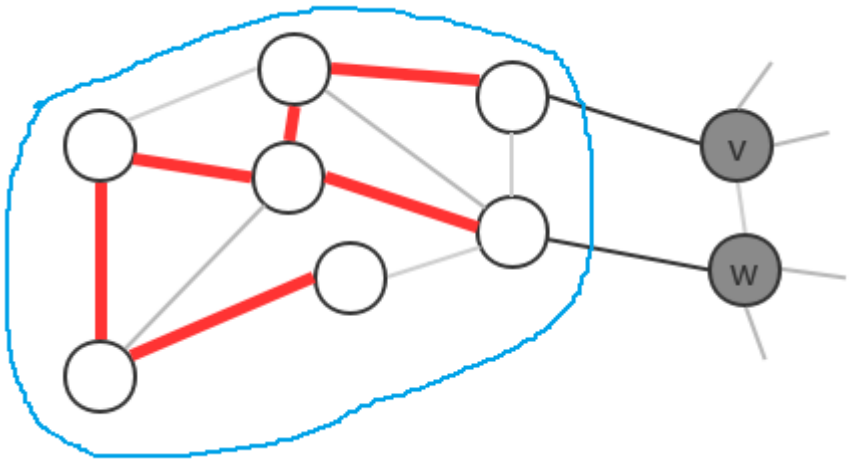
计算图的最小生成树的算法有很多种，但这些算法都可以看做是贪心算法的一种特殊情况，这些算法的不同之处在于保存切分和判定权重最小的横切边的方式。

4.4 Prim算法

我们学习第一种计算最小生成树的方法叫Prim算法，它的每一步都会为一棵生成中的树添加一条边。一开始这棵树只有一个顶点，然后会向它添加V-1条边，每次总是将下一条连接树中的顶点与不在树中的顶点且权重最小的边加入到树中。

Prim算法的切分规则：

把最小生成树中的顶点看做是一个集合，把不在最小生成树中的顶点看做是另外一个集合。



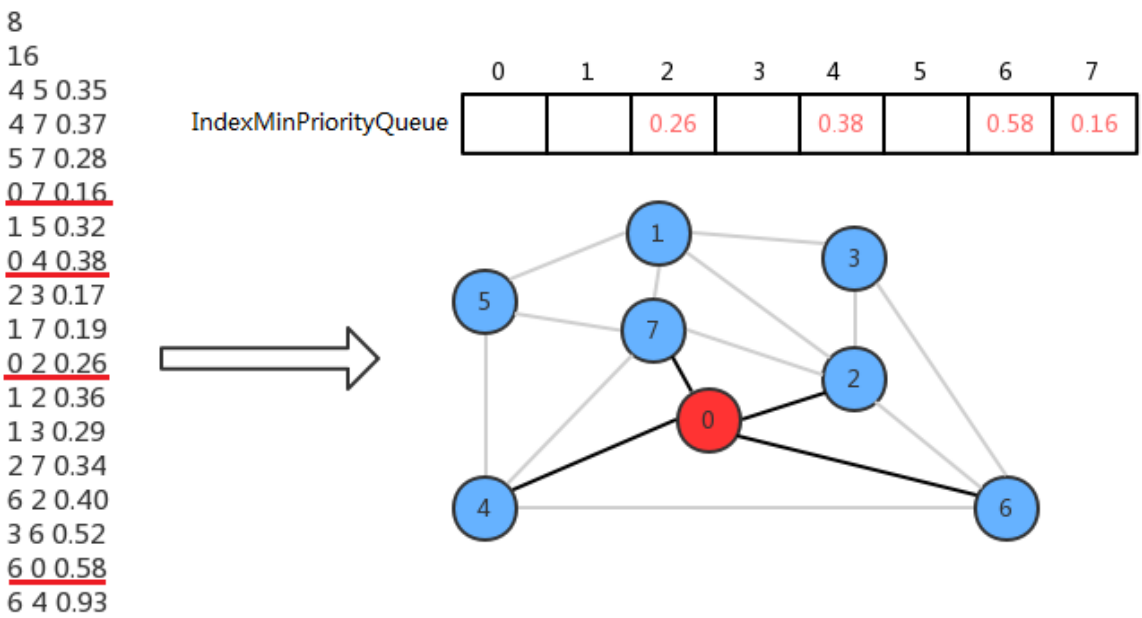
4.4.1 Prim算法API设计

类名	PrimMST
构造方法	PrimMST(EdgeWeightedGraph G)：根据一副加权无向图，创建最小生成树计算对象；
成员方法	1.private void visit(EdgeWeightedGraph G, int v)：将顶点v添加到最小生成树中，并且更新数据 2.public Queue edges():获取最小生成树的所有边
成员变量	1.private Edge[] edgeTo: 索引代表顶点，值表示当前顶点和最小生成树之间的最短边 2.private double[] distTo: 索引代表顶点，值表示当前顶点和最小生成树之间的最短边的权重 3.private boolean[] marked:索引代表顶点，如果当前顶点已经在树中，则值为true，否则为false 4.private IndexMinPriorityQueue pq:存放树中顶点与非树中顶点之间的有效横切边

4.4.2 Prim算法的实现原理

Prim算法始终将图中的顶点切分成两个集合，最小生成树顶点和非最小生成树顶点，通过不断的重复做某些操作，可以逐渐将非最小生成树中的顶点加入到最小生成树中，直到所有的顶点都加入到最小生成树中。

我们在设计API的时候，使用最小索引优先队列存放树中顶点与非树中顶点的有效横切边，那么它是如何表示的呢？我们可以让最小索引优先队列的索引值表示图的顶点，让最小索引优先队列中的值表示从其他某个顶点到当前顶点的边权重。



初始化状态，先默认0是最小生成树中的唯一顶点，其他的顶点都不在最小生成树中，此时横切边就是顶点0的邻接表中0-2,0-4,0-6,0-7这四条边，我们只需要将索引优先队列的2、4、6、7索引处分别存储这些边的权重值就可以表示了。

现在只需要从这四条横切边中找出权重最小的边，然后把对应的顶点加进来即可。所以找到0-7这条横切边的权重最小，因此把0-7这条边添加进来，此时0和7属于最小生成树的顶点，其他的不属于，现在顶点7的邻接表中的边也成为了横切边，这时需要做两个操作：

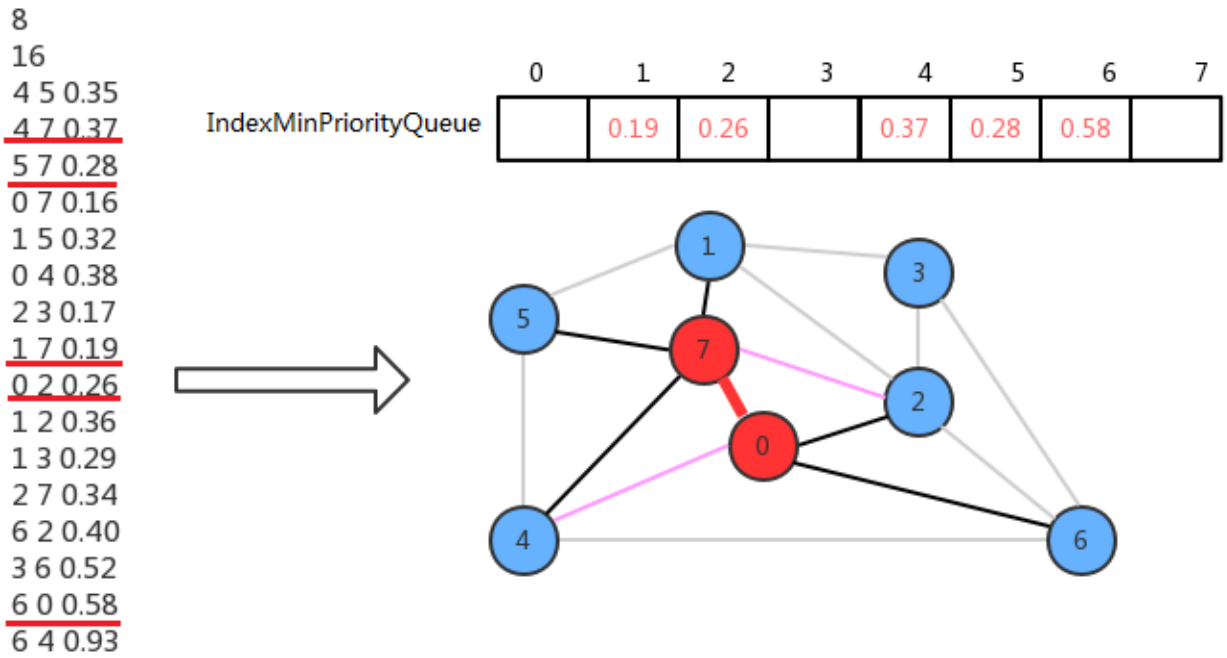
1、0-7这条边已经不是横切边了，需要让它失效：

只需要调用最小索引优先队列的delMin()方法即可完成；

2、2和4顶点各有两条连接指向最小生成树，需要只保留一条：

4-7的权重小于0-4的权重，所以保留4-7，调用索引优先队列的change(4,0.37)即可，

0-2的权重小于2-7的权重，所以保留0-2，不需要做额外操作。



我们不断重复上面的动作，就可以把所有的顶点添加到最小生成树中。

4.4.3 代码

```
package cn.itcast;

public class PrimMST {
    //索引代表顶点，值表示当前顶点和最小生成树之间的最短边
    private Edge[] edgeTo;
    //索引代表顶点，值表示当前顶点和最小生成树之间的最短边的权重
    private double[] distTo;
    //索引代表顶点，如果当前顶点已经在树中，则值为true，否则为false
    private boolean[] marked;
    //存放树中顶点与非树中顶点之间的有效横切边
    private IndexMinPriorityQueue<Double> pq;

    //根据一副加权无向图，创建最小生成树计算对象
    public PrimMST(EdgeWeightedGraph G) {
        //创建一个和图的顶点数一样大小的Edge数组，表示边
        this.edgeTo = new Edge[G.V()];
```

```
        //创建一个和图的顶点数一样大小的double数组，表示权重，并且初始化数组中的内容为无穷大，无穷大即表示不存在这样的边
        this.distTo = new double[G.V()];
        for (int i = 0; i < distTo.length; i++) {
            distTo[i] = Double.POSITIVE_INFINITY;
        }
        //创建一个和图的顶点数一样大小的boolean数组，表示当前顶点是否已经在树中
        this.marked = new boolean[G.V()];
        //创建一个和图的顶点数一样大小的索引优先队列，存储有效横切边
        this.pq = new IndexMinPriorityQueue<>(G.V());

        //默认让顶点0进入树中，但0顶点目前没有与树中其他的顶点相连接，因此初始化distTo[0]=0.0
        distTo[0] = 0.0;
        //使用顶点0和权重0初始化pq
        pq.insert(0, 0.0);
        //遍历有效边队列
        while (!pq.isEmpty()) {
            //找到权重最小的横切边对应的顶点，加入到最小生成树中
            visit(G, pq.delMin());
        }
    }


    //将顶点v添加到最小生成树中，并且更新数据
    private void visit(EdgeWeightedGraph G, int v) {
        //把顶点v添加到树中
        marked[v] = true;
        //遍历顶点v的邻接表,得到每一条边Edge e,
        for (Edge e : G.adj(v)) {
            //边e的一个顶点是v，找到另外一个顶点w；
            int w = e.other(v);
            //检测是否已经在树中，如果在，则继续下一次循环，如果不在，则需要修正当前顶点w距离最小生成树的最小边edgeTo[w]以及它的权重distTo[w]，还有有效横切边也需要修正
            if (marked[w]) {
                continue;
            }

            //如果v-w边e的权重比目前distTo[w]权重小，则需要修正数据
            if (e.weight() < distTo[w]) {
                //把顶点w距离最小生成树的边修改为e
                edgeTo[w] = e;
                //把顶点w距离最小生成树的边的权重修改为e.weight()
                distTo[w] = e.weight();
                //如果pq中存储的有效横切边已经包含了w顶点，则需要修正最小索引优先队列w索引关联的权重值
                if (pq.contains(w)) {
                    pq.changeItem(w, e.weight());
                } else {
                    //如果pq中存储的有效横切边不包含w顶点，则需要向最小索引优先队列中添加v-w和其权重值
                    pq.insert(w, e.weight());
                }

            }
```

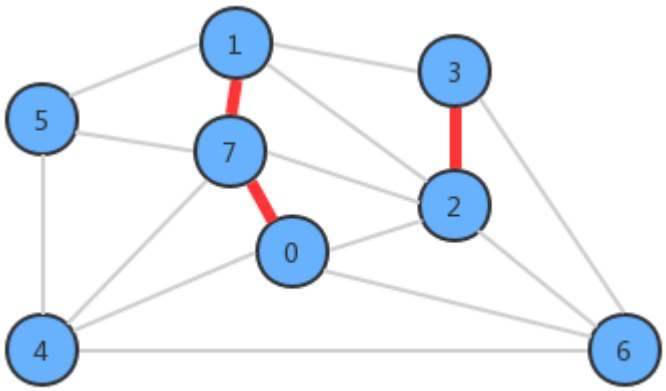
```
        }
    }

    //获取最小生成树的所有边
    public Queue<Edge> edges() {
        //创建队列
        Queue<Edge> edges = new Queue<>();
        //遍历edgeTo数组，找到每一条边，添加到队列中
        for (int i = 0; i < marked.length; i++) {
            if (edgeTo[i]!=null){
                edges.enqueue(edgeTo[i]);
            }
        }
        return edges;
    }
}


//测试代码
public class PrimTest {
    public static void main(String[] args) throws Exception {
        //创建输入流
        BufferedReader reader = new BufferedReader(new
InputStreamReader(PrimTest.class.getClassLoader().getResourceAsStream("min_create_tree_test
.txt")));
        //读取顶点数目，初始化EdgeWeightedGraph图
        int number = Integer.parseInt(reader.readLine());
        EdgeWeightedGraph G = new EdgeWeightedGraph(number);
        //读取边的数目
        int edgeNumber = Integer.parseInt(reader.readLine());
        //循环读取每一条边，并调用addEdge方法
        for (int i = 0; i < edgeNumber; i++) {
            String line = reader.readLine();
            int v = Integer.parseInt(line.split(" ")[0]);
            int w = Integer.parseInt(line.split(" ")[1]);
            double weight = Double.parseDouble(line.split(" ")[2]);
            G.addEdge(new Edge(v, w, weight));
        }

        //构建PrimMST对象
        PrimMST mst = new PrimMST(G);
        //获取最小生成树的边
        Queue<Edge> edges = mst.edges();
        //打印输出
        for (Edge edge : edges) {
            if (edge!=null){
                System.out.println(edge.either() + "-" + edge.other(edge.either()) + "::" +
edge.weight());
            }
        }
    }
}
```

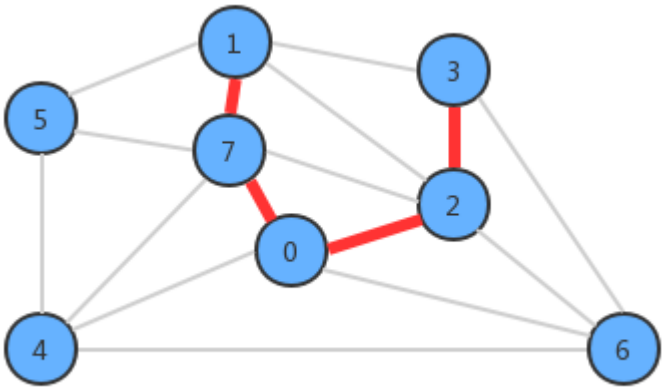
4.5 kruskal算法

kruskal算法是计算一副加权无向图的最小生成树的另外一种算法，它的主要思想是按照边的权重(从小到大)处理它们，将边加入最小生成树中，加入的边不会与已经加入最小生成树的边构成环，直到树中含有V-1条边为止。

kruskal算法和prim算法的区别：

Prim算法是一条边一条边的构造最小生成树，每一步都为一棵树添加一条边。kruskal算法构造最小生成树的时候也是一条边一条边地构造，但它的切分规则是不一样的。它每一次寻找的边会连接一片森林中的两棵树。如果一副加权无向图由V个顶点组成，初始化情况下每个顶点都构成一棵独立的树，则V个顶点对应V棵树，组成一片森林，kruskal算法每一次处理都会将两棵树合并为一棵树，直到整个森林中只剩一棵树为止。



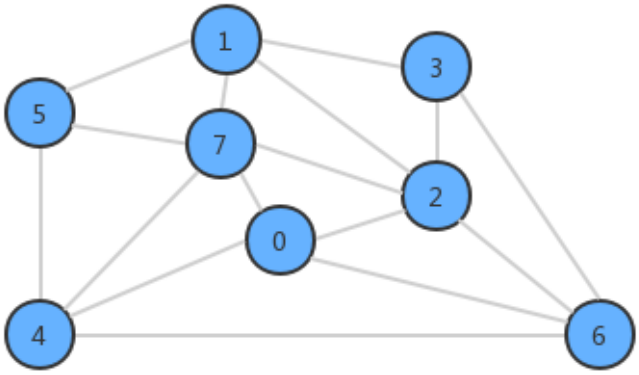


4.5.1 kruskal算法API设计

类名	**KruskalMST**
构造方法	KruskalMST(EdgeWeightedGraph G)：根据一副加权无向图，创建最小生成树计算对象；
成员方法	1.public Queue edges():获取最小生成树的所有边
成员变量	1.private Queue mst：保存最小生成树的所有边 2.private UF_Tree_Weighted uf: 索引代表顶点，使用uf.connect(v,w)可以判断顶点v和顶点w是否在同一颗树中，使用uf.union(v,w)可以把顶点v所在的树和顶点w所在的树合并 3.private MinPriorityQueue pq: 存储图中所有的边，使用最小优先队列，对边按照权重进行排序

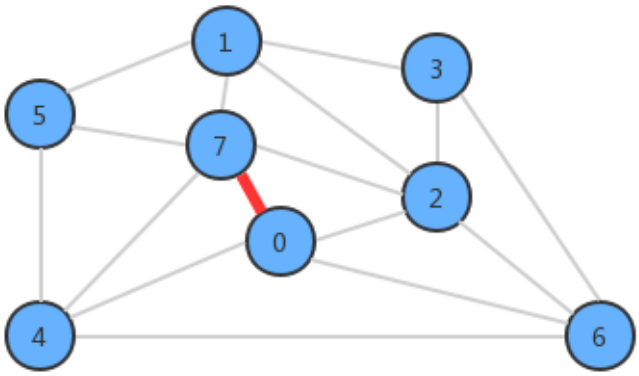
4.5.2 kruskal算法的实现原理

在设计API的时候，使用了一个MinPriorityQueue pq存储图中所有的边，每次使用pq.delMin()取出权重最小的边，并得到该边关联的两个顶点v和w，通过uf.connect(v,w)判断v和w是否已经连通，如果连通，则证明这两个顶点在同一棵树中，那么就不能再把这条边添加到最小生成树中，因为在一棵树的任意两个顶点上添加一条边，都会形成环，而最小生成树不能有环的存在，如果不连通，则通过uf.connect(v,w)把顶点v所在的树和顶点w所在的树合并成一棵树，并把这条边加入到mst队列中，这样如果把所有的边处理完，最终mst中存储的就是最小生树的所有边。



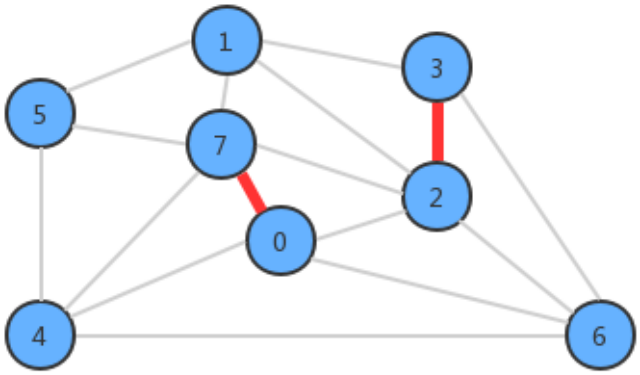
pq 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35
2 7 0.34
1 5 0.32
1 3 0.29
5 7 0.28
0 2 0.26
1 7 0.19
2 3 0.17
0 7 0.16



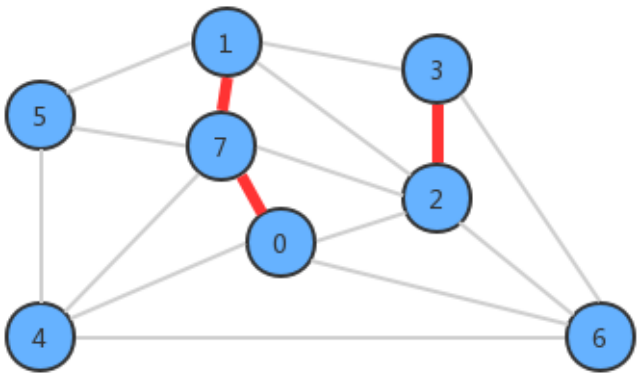
pq 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35
2 7 0.34
1 5 0.32
1 3 0.29
5 7 0.28
0 2 0.26
1 7 0.19
2 3 0.17



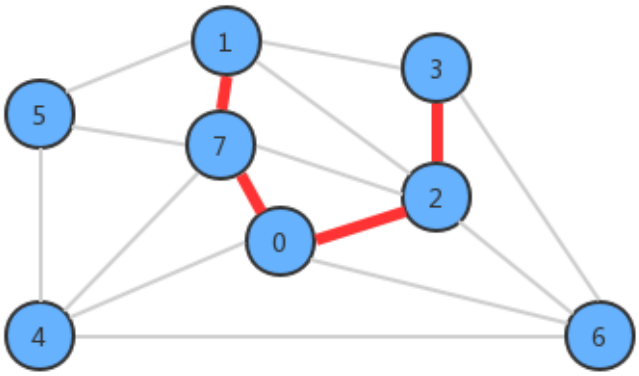
pq 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35
2 7 0.34
1 5 0.32
1 3 0.29
5 7 0.28
0 2 0.26
1 7 0.19



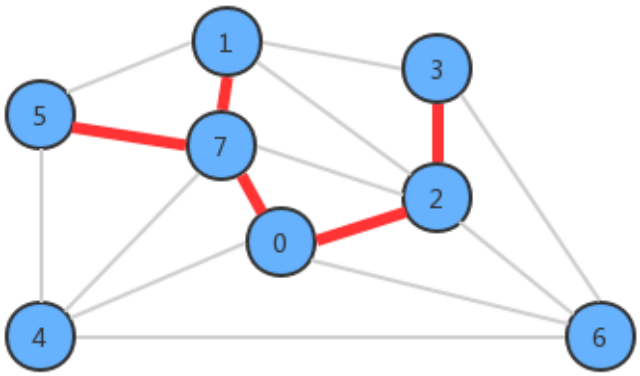
pq 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35
2 7 0.34
1 5 0.32
1 3 0.29
5 7 0.28
0 2 0.26



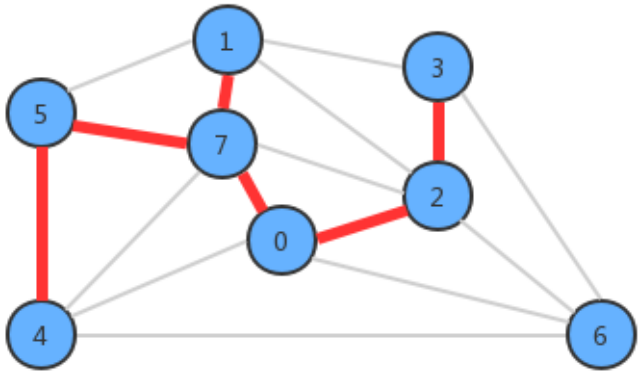
pq 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35
2 7 0.34
1 5 0.32
1 3 0.29
5 7 0.28



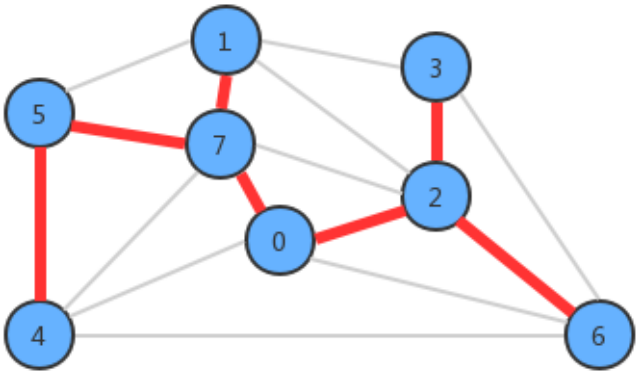
MinPQ 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4
0 4 0.38
4 7 0.37
1 2 0.36
4 5 0.35



MinPQ 中的边按照权重排序

6 4 0.93
6 0 0.58
3 6 0.52
6 2 0.4



4.5.3 代码

```
1  public class KruskalMST {
2      //保存最小生成树的所有边
3      private Queue<Edge> mst;
4      //索引代表顶点，使用uf.connect(v,w)可以判断顶点v和顶点w是否在同一颗树中，使用uf.union(v,w)可以把顶点v所在的树和顶点w所在的树合并
5      private UF_Tree_Weighted uf;
6      //存储图中所有的边，使用最小优先队列，对边按照权重进行排序
7      private MinPriorityQueue<Edge> pq;
8
9      //根据一副加权无向图，创建最小生成树计算对象
10     public KruskalMST(EdgeWeightedGraph G) {
11         //初始化mst队列
12         this.mst = new Queue<Edge>();
13         //初始化并查集对象uf,容量和图的顶点数相同
14         this.uf = new UF_Tree_Weighted(G.V());
15         //初始化最小优先队列pq，容量比图的边的数量大1，并把图中所有的边放入pq中
16         this.pq = new MinPriorityQueue<>(G.E()+1);
17         for (Edge edge : G.edges()) {
18             pq.insert(edge);
19         }
20
21         //如果优先队列pq不为空，也就是还有边未处理，并且mst中的边还不到V-1条，继续遍历
```

```
        while (!pq.isEmpty() && mst.size() < G.V() - 1) {
            //取出pq中权重最小的边e
            Edge e = pq.delMin();
            //获取边e的两个顶点v和w
            int v = e.either();
            int w = e.other(v);
            /*
            通过uf.connect(v,w)判断v和w是否已经连通，
                如果连通:
                    则证明这两个顶点在同一棵树中，那么就不能再把这条边添加到最小生成树中，因为在一棵树的任意两个顶点上添加一条边，都会形成环，
                    而最小生成树不能有环的存在;
                如果不连通:
                    则通过uf.connect(v,w)把顶点v所在的树和顶点w所在的树合并成一棵树,并把这条边加入到mst队列中
            */
            if (uf.connected(v,w)){
                continue;
            }

            uf.union(v,w);
            mst.enqueue(e);
        }
    }

    //获取最小生成树的所有边
    public Queue<Edge> edges() {
        return mst;
    }
}

//测试代码
public class KruskalTest {
    public static void main(String[] args) throws Exception {
        //创建输入流
        BufferedReader reader = new BufferedReader(new InputStreamReader(KruskalTest.class.getClassLoader().getResourceAsStream("min_create_tree_test.txt")));
        //读取顶点数目，初始化EdgeWeightedGraph图
        int number = Integer.parseInt(reader.readLine());
        EdgeWeightedGraph G = new EdgeWeightedGraph(number);
        //读取边的数目
        int edgeNumber = Integer.parseInt(reader.readLine());
        //循环读取每一条边，并调用addEdge方法
        for (int i = 0; i < edgeNumber; i++) {
            String line = reader.readLine();
            int v = Integer.parseInt(line.split(" ")[0]);
            int w = Integer.parseInt(line.split(" ")[1]);
            double weight = Double.parseDouble(line.split(" ")[2]);
            G.addEdge(new Edge(v, w, weight));
        }


        //构建PrimMST对象
```

```
        KruskalMST mst = new KruskalMST(G);
        //获取最小生成树的边
        Queue<Edge> edges = mst.edges();
        //打印输出
        for (Edge edge : edges) {
            if (edge!=null){
                System.out.println(edge.either() + "-" + edge.other(edge.either()) + "::" +
edge.weight());
            }
        }
    }
}
```

五、加权有向图

之前学习的加权无向图中，边是没有方向的，并且同一条边会同时出现在该边的两个顶点的邻接表中，为了能够处理含有方向性的图的问题，我们需要实现以下加权有向图。

5.1加权有向图边的表示

API设计：

类名	DirectedEdge
构造方法	DirectedEdge(int v,int w,double weight)：通过顶点v和w，以及权重weight值构造一个边对象
成员方法	1.public double weight():获取边的权重值 2.public int from():获取有向边的起点 3.public int to():获取有向边的终点
成员变量	1.private final int v：起点 2.private final int w：终点 3.private final double weight：当前边的权重

代码：

```
public class DirectedEdge {
    private final int v;//起点
    private final int w;//终点
    private final double weight;//当前边的权重

    //通过顶点v和w，以及权重weight值构造一个边对象
    public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

```

```
    //获取边的权重值
    public double weight(){
        return weight;
    }

    //获取有向边的起点
    public int from(){
      return v;
    }

    //获取有向边的终点
    public int to(){
       return w;
    }
}
```

5.2 加权有向图的实现

之前我们已经完成了有向图，在有向图的基础上，我们只需要把边的表示切换成DirectedEdge对象即可。

API设计：

类名	EdgeWeightedDigraph
构造方法	EdgeWeightedDigraph(int V)：创建一个含有V个顶点的空加权有向图
成员方法	1.public int V():获取图中顶点的数量 2.public int E():获取图中边的数量 3.public void addEdge(DirectedEdge e):向加权有向图中添加一条边e 4.public Queue adj(int v)：获取由顶点v指出的所有的边 5.public Queue edges()：获取加权有向图的所有边
成员变量	1.private final int V: 记录顶点数量 2.private int E: 记录边数量 3.private Queue[] adj: 邻接表

代码：

```
public class EdgeWeightedDigraph {
    //顶点总数
    private final int V;
    //边的总数
    private int E;
    //邻接表
    private Queue<DirectedEdge>[] adj;

    //创建一个含有V个顶点的空加权有向图
    public EdgeWeightedDigraph(int V) {
        //初始化顶点数量

        this.V = V;
```

```
        //初始化边的数量
        this.E = 0;
        //初始化邻接表
        this.adj = new Queue[V];
        //初始化邻接表中的空队列
        for (int i = 0; i < adj.length; i++) {
            adj[i] = new Queue<DirectedEdge>();
        }

    }

    //获取图中顶点的数量
    public int V() {
        return V;
    }

    //获取图中边的数量
    public int E() {
        return E;
    }


    //向加权有向图中添加一条边e
    public void addEdge(DirectedEdge e) {
        //获取有向边的起点
        int v = e.from();
        //因为是有向图，所以边e只需要出现在起点v的邻接表中
        adj[v].enqueue(e);
        //边的数量+1
        E++;
    }

    //获取由顶点v指出的所有的边
    public Queue<DirectedEdge> adj(int v) {
        return adj[v];
    }

    //获取加权有向图的所有边
    public Queue<DirectedEdge> edges() {
        //创建一个队列，存储所有的边
        Queue<DirectedEdge> allEdge = new Queue<>();
        //遍历顶点，拿到每个顶点的邻接表
        for (int v = 0; v < this.V; v++) {
            //遍历邻接表，拿到邻接表中的每条边存储到队列中
            for (DirectedEdge e : adj(v)) {
                allEdge.enqueue(e);
            }
        }
        return allEdge;
    }
}
```

六、最短路径

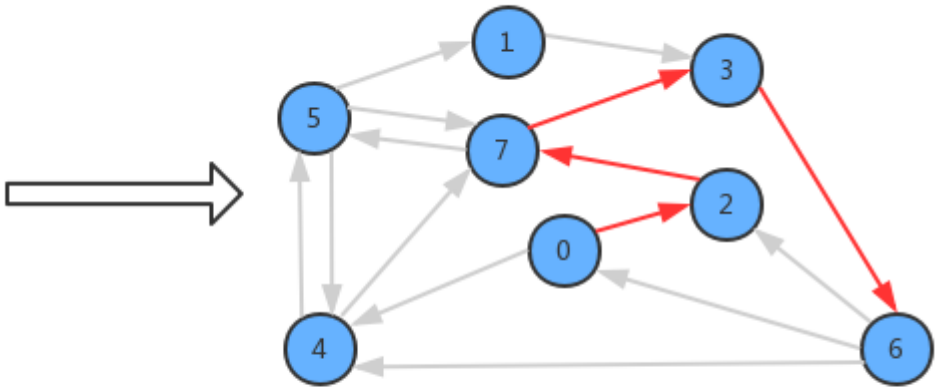
有了加权有向图之后，我们立刻就能联想到实际生活中的使用场景，例如在一副地图中，找到顶点a与地点b之间的路径，这条路径可以是距离最短，也可以是时间最短，也可以是费用最小等，如果我们把 `距离/时间/费用` 看做是成本，那么就需要找到地点a和地点b之间成本最小的路径，也就是我们接下来要解决的最短路径问题。

6.1 最短路径定义及性质

定义：

在一副加权有向图中，从顶点s到顶点t的最短路径是所有从顶点s到顶点t的路径中总权重最小的那条路径。

```
8
15
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



从顶点0到顶点6的最短路径

性质：

1.路径具有方向性；

2.权重不一定等价于距离。权重可以是距离、时间、花费等内容，权重最小指的是成本最低

3.只考虑连通图。一副图中并不是所有的顶点都是可达的，如果s和t不可达，那么它们之间也就不存在最短路径，为了简化问题，这里只考虑连通图。

4.最短路径不一定是唯一的。从一个顶点到达另外一个顶点的权重最小的路径可能会有很多条，这里只需要找出一条即可。

最短路径树：

给定一副加权有向图和一个顶点s，以s为起点的一棵最短路径树是图的一副子图，它包含顶点s以及从s可达的所有顶点。这棵有向树的根结点为s，树的每条路径都是有向图中的一条最短路径。

6.2 最短路径树API设计

计算最短路径树的经典算法是dijstra算法，为了实现它，先设计如下API：

类名	DijkstraSP
构造方法	public DijkstraSP(EdgeWeightedDigraph G, int s)：根据一副加权有向图G和顶点s，创建一个计算顶点为s的最短路径树对象
成员方法	1.private void relax(EdgeWeightedDigraph G, int v)：松弛图G中的顶点v 2.public double distTo(int v):获取从顶点s到顶点v的最短路径的总权重 3.public boolean hasPathTo(int v):判断从顶点s到顶点v是否可达 4.public Queue pathTo(int v):查询从起点s到顶点v的最短路径中所有的边
成员变量	1.private DirectedEdge[] edgeTo: 索引代表顶点，值表示从顶点s到当前顶点的最短路径上的最后一条边 2.private double[] distTo: 索引代表顶点，值从顶点s到当前顶点的最短路径的总权重 3.private IndexMinPriorityQueue pq:存放树中顶点与非树中顶点之间的有效横切边

6.3 松弛技术

松弛这个词来源于生活：一条橡皮筋沿着两个顶点的某条路径紧紧展开，如果这两个顶点之间的路径不止一条，还有存在更短的路径，那么把皮筋转移到更短的路径上，皮筋就可以放松了。

松弛这种简单的原理刚好可以用来计算最短路径树。

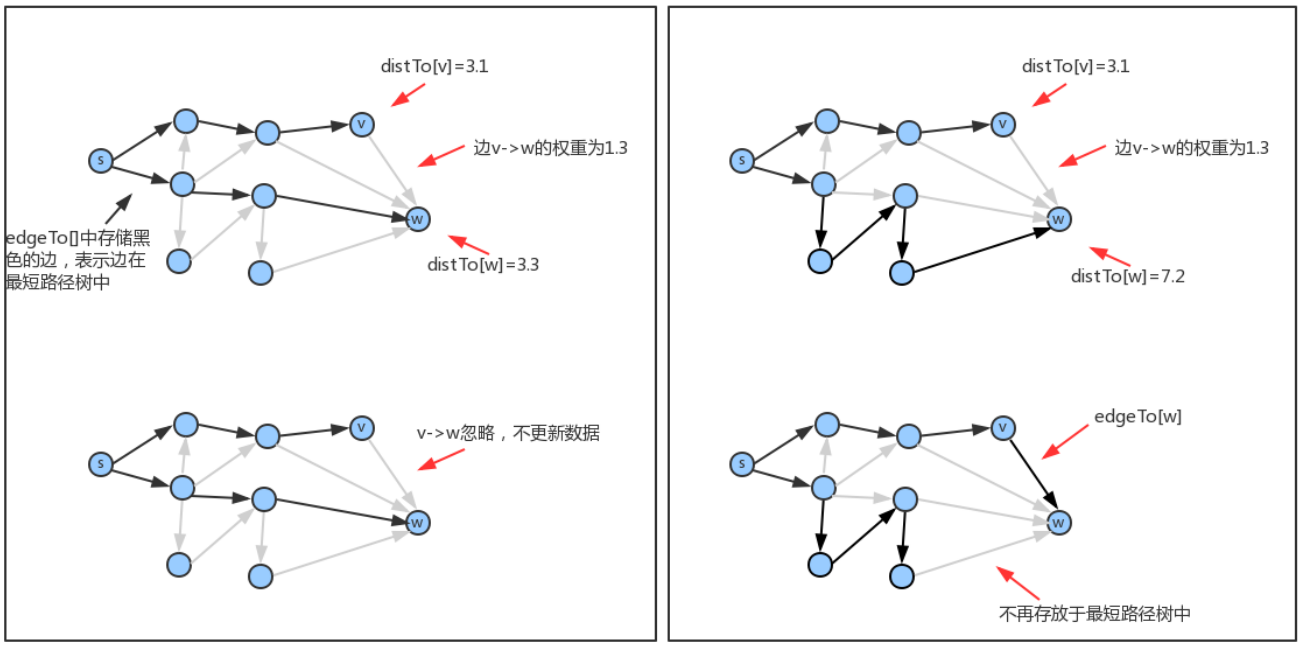
在我们的API中，需要用到两个成员变量edgeTo和distTo，分别存储边和权重。一开始给定一幅图G和顶点s，我们只知道图的边以及这些边的权重，其他的一无所知，此时初始化顶点s到顶点s的最短路径的总权重disto[s]=0；顶点s到其他顶点的总权重默认为无穷大，随着算法的执行，不断的使用松弛技术处理图的边和顶点，并按一定的条件更新edgeTo和distTo中的数据，最终就可以得到最短路劲树。

边的松弛：

放松边v->w意味着检查从s到w的最短路径是否先从s到v，然后再从v到w？

如果是，则v-w这条边需要加入到最短路径树中，更新edgeTo和distTo中的内容：edgeTo[w]=表示v->w这条边的DirectedEdge对象，distTo[w]=distTo[v]+v->w这条边的权重；

如果不是，则忽略v->w这条边。

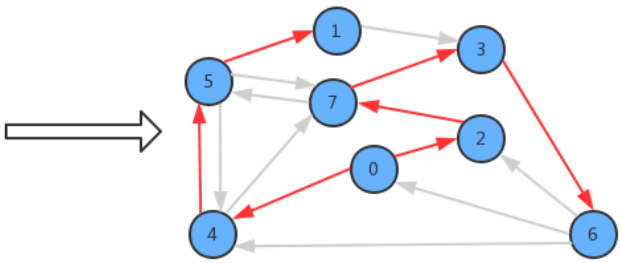


顶点的松弛：

顶点的松弛是基于边的松弛完成的，只需要把某个顶点指出的所有边松弛，那么该顶点就松弛完毕。例如要松弛顶点v，只需要遍历v的邻接表，把每一条边都松弛，那么顶点v就松弛了。

如果把起点设置为顶点0，那么找出起点0到顶点6的最短路径0->2->7>3->6的过程如下:

```
8
15
4->5 0.35
5->4 0.35
4->7 0.37
5->7 0.28
7->5 0.28
5->1 0.32
0->4 0.38
0->2 0.26
7->3 0.39
1->3 0.29
2->7 0.34
6->2 0.40
3->6 0.52
6->0 0.58
6->4 0.93
```



v	edgeTo[]
0	null
1	5->1
2	0->2
3	7->3
4	0->4
5	4->5
6	3->6
7	2->7

pathTo(6)

```
3->6
3->6 7->3
3->6 7->3 2->7
3->6 7->3 2->7 0->2
```

6.4 Dijstra算法实现

Disjstra算法的实现和Prim算法很类似，构造最短路径树的每一步都是向这棵树中添加一条新的边，而这条新的边是有效横切边pq队列中的权重最小的边。

```
1  public class DijkstraSP {
2      //索引代表顶点，值表示从顶点s到当前顶点的最短路径上的最后一条边
3      private DirectedEdge[] edgeTo;
4      //索引代表顶点，值从顶点s到当前顶点的最短路径的总权重
5      private double[] distTo;
```

```java
    //存放树中顶点与非树中顶点之间的有效横切边
    private IndexMinPriorityQueue<Double> pq;

    //根据一副加权有向图G和顶点s，创建一个计算顶点为s的最短路径树对象
    public DijkstraSP(EdgeWeightedDigraph G, int s){
        //创建一个和图的顶点数一样大小的DirectedEdge数组，表示边
        this.edgeTo = new DirectedEdge[G.V()];
        //创建一个和图的顶点数一样大小的double数组，表示权重，并且初始化数组中的内容为无穷大，无穷大即表示不存在这样的边
        this.distTo = new double[G.V()];
        for (int i = 0; i < distTo.length; i++) {
            distTo[i] = Double.POSITIVE_INFINITY;
        }
        //创建一个和图的顶点数一样大小的索引优先队列，存储有效横切边
        this.pq = new IndexMinPriorityQueue<>(G.V());
        //默认让顶点s进入树中，但s顶点目前没有与树中其他的顶点相连接，因此初始化distTo[s]=0.0
        distTo[s] = 0.0;
        //使用顶点s和权重0.0初始化pq
        pq.insert(s, 0.0);
        //遍历有效边队列
        while (!pq.isEmpty()) {
            //松弛图G中的顶点
            relax(G, pq.delMin());
        }
    }

    //松弛图G中的顶点v
    private void relax(EdgeWeightedDigraph G, int v){
        //松弛顶点v就是松弛顶点v邻接表中的每一条边，遍历邻接表
        for (DirectedEdge e : G.adj(v)) {
            //获取边e的终点
            int w = e.to();
            //起点s到顶点w的权重是否大于起点s到顶点v的权重+边e的权重,如果大于，则修改s->w的路径：edgeTo[w]=e,并修改distTo[v] = distTo[v]+e.weitht(),如果不大于，则忽略
            if (distTo(w)>distTo(v)+e.weight()){
                distTo[w]=distTo[v]+e.weight();
                edgeTo[w]=e;

                //如果顶点w已经存在于优先队列pq中，则重置顶点w的权重
                if (pq.contains(w)){
                    pq.changeItem(w,distTo(w));
                }else{
                    //如果顶点w没有出现在优先队列pq中，则把顶点w及其权重加入到pq中
                    pq.insert(w,distTo(w));
                }

            }
        }
    }

    //获取从顶点s到顶点v的最短路径的总权重
    public double distTo(int v){

        return distTo[v];
```

```
    }

    //判断从顶点s到顶点v是否可达
    public boolean hasPathTo(int v){
        return distTo[v]<Double.POSITIVE_INFINITY;
    }

    //查询从起点s到顶点v的最短路径中所有的边
    public Queue<DirectedEdge> pathTo(int v){
        //如果顶点s到v不可达，则返回null
        if (!hasPathTo(v)){
            return null;
        }
        //创建队列Queue保存最短路径的边
        Queue<DirectedEdge> edges = new Queue<>();
        //从顶点v开始，逆向寻找，一直找到顶点s为止，而起点s为最短路劲树的根结点，所以edgeTo[s]=null;
        DirectedEdge e=null;
        while(true){
            e = edgeTo[v];
            if (e==null){
                break;
            }
            edges.enqueue(e);
            v = e.from();
        }
        return edges;

    }

}

//测试代码
public class DijkstraSpTest {
    public static void main(String[] args) throws Exception {
        //创建输入流
        BufferedReader reader = new BufferedReader(new InputStreamReader(DijkstraSpTest.class.getClassLoader().getResourceAsStream("min_route_test.txt")));
        //读取顶点数目，初始化EdgeWeightedDigraph图
        int number = Integer.parseInt(reader.readLine());
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(number);
        //读取边的数目
        int edgeNumber = Integer.parseInt(reader.readLine());
        //循环读取每一条边，并调用addEdge方法
        for (int i = 0; i < edgeNumber; i++) {
            String line = reader.readLine();
            int v = Integer.parseInt(line.split(" ")[0]);
            int w = Integer.parseInt(line.split(" ")[1]);
            double weight = Double.parseDouble(line.split(" ")[2]);
            G.addEdge(new DirectedEdge(v, w, weight));
        }


```

```
        //根据图G和顶点0，构建DijkstraSP对象
        DijkstraSP dsp = new DijkstraSP(G, 0);
        //获取起点0到顶点6的最短路径
        Queue<DirectedEdge> edges = dsp.pathTo(6);
        //打印输出
        for (DirectedEdge edge : edges) {
            System.out.println(edge.from() + "->" + edge.to() + "::" + edge.weight());
        }
    }
}
```