

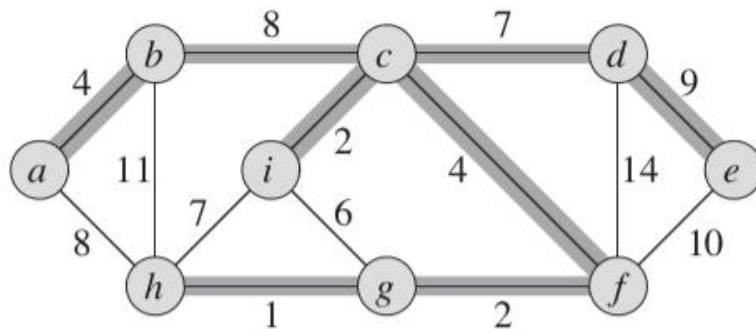
简介

最近几天在家闲来无事，就打算把几个原来一直不太清楚的问题整理一下。现在代码都写得想吐了，好在问题也弄清楚了。Minimum Spanning Tree是一个图算法中很典型的应用，我们常用的构建这种树的算法有两种，Prim算法和Kruskal算法。从表面上看起来这两种方法比较复杂，等摸清楚了他们的思路之后会发现还是基于一个很简单的思想。另外，在实现各种细节的时候所面临的各种问题，包括怎么对图的结构进行定义，怎么来保存和返回各种数据类型，这些都构成一个很有意思的解决方法。当我们将问题一个个的击破之后，感觉最后的解法像是搭积木一样好玩。

定义和结构

Minimum Spanning Tree是一棵树。但是它是基于图来构造的。这里基于一个基本的前提条件就是这个图本身是连通的。在这个基础上，假设我们要构造一棵树，使得它们不仅连接了所有的节点，而且它们所有边的权值之和是最小的。而通过什么样的方式找到这些边和节点，就是我们需要考虑的问题。

一个典型的Minimum Spanning Tree如下图：



在图中加粗的线条将所有的点都连接在一起，同时它们的总权值为最小。

在考虑各种找寻的算法之前，我们肯定要考虑一下用什么样的结构来描述这种图。我前面的一篇[文章](#)描述过图的结构和遍历思路。不过在那里只是针对节点和边之间的关系做了一个描述，并没有考虑到如果每一个边有对应权值的情况。所以我们需要对原有定义的结构做一点调整。回想原来的结构里，我们对所有节点编码为数字0---n-1。而所有的边则作为每一个链表里对应的数值。这里，我们需要定义一个Edge对象。假定我们这里仅考虑无向图的情况，那么对于每个Edge对象，我们需要将它同时加入到对应到两个节点的adj链表里。

这样，我们可以先定义Edge对象如下：

Java代码 ☆

```
public class Edge implements Comparable<Edge> {
    private final int v;
    private final int w;
    private final double weight;

    public Edge(int v, int w, double weight) {
        if(v < 0)
            throw new IndexOutOfBoundsException(
                "Vertex name must be a nonnegative integer");
        if(w < 0)
            throw new IndexOutOfBoundsException(
                "Vertex name must be a nonnegative integer");
        if(Double.isNaN(weight))
            throw new IllegalArgumentException("Weight is NaN");
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public double getWeight() {
        return weight;
    }

    public int either() {
        return v;
    }

    public int other(int vertex) {
        if(vertex == v)
            return w;
        else if(vertex == w)
            return v;
        else
            throw new RuntimeException("Inconsistent edge");
    }
}
```

```


@Override
public int compareTo(Edge that) {
    if(this.getWeight() < that.getWeight())
        return -1;
    else if(this.getWeight() > that.getWeight())
        return 1;
    else
        return 0;
}

@Override
public String toString() {
    return String.format("%d-%d %.2f", v, w, weight);
}
}

```

我们定义的这个Edge类有几个需要注意的地方。一方面它是连接两个节点，对于无向图来说，无所谓谁先谁后，只需要标注好一个是哪个节点，另外一个一个是哪个节点就好办了。另外，因为既然是定义成一个边，它有自己的权值，这些Edge对象可能会作各种比较。尤其在后面一些算法里要取一些权值低的边。要使得Edge对象支持比较，我们还需要实现接口Comparable。

增加了Edge类之后，我们需要的是在原来图定义的基础上修改一下。原来节点的adj关联表是Integer，这里则是Edge对象。那么这个图的典型构造函数如下：


Java代码 

```

public EdgeWeightedGraph(int vertices) {
    if(vertices < 0)
        throw new IllegalArgumentException(
            "Number of vertices must be nonnegative");
    this.vertices = vertices;
    this.edges = 0;
    adj = new ArrayList<LinkedList<Edge>>();
    for(int i = 0; i < vertices; i++) {
        adj.add(new LinkedList<Edge>());
    }
}

```

在给定长度的参数基础上，我们创建一个List，里面的每个成员是一个链表。如果我们要增加一个边到集合里，或者返回所有的边，我们需要定义addEdge和edges方法。它们的定义如下：

Java代码 

```

public void addEdge(Edge e) {
    int v = e.either();
    int w = e.other(v);
    if(v < 0 || v >= vertices)
        throw new IndexOutOfBoundsException(
            "vertex " + v + " is not between 0 and " + (vertices - 1));
    if(w < 0 || w >= vertices)
        throw new IndexOutOfBoundsException(
            "vertex " + w + " is not between 0 and " + (vertices - 1));
    adj.get(v).add(e);
    adj.get(w).add(e);
    edges++;
}

public LinkedList<Edge> edges() {
    LinkedList<Edge> list = new LinkedList<Edge>();
    for(int v = 0; v < vertices; v++) {
        int selfLoops = 0;
        for(Edge e : adj(v)) {
            if(e.other(v) > v) {
                list.add(e);
            } else if(e.other(v) == v) {
                if(selfLoops % 2 == 0) list.add(e);
                selfLoops++;
            }
        }
    }
    return list;
}

```

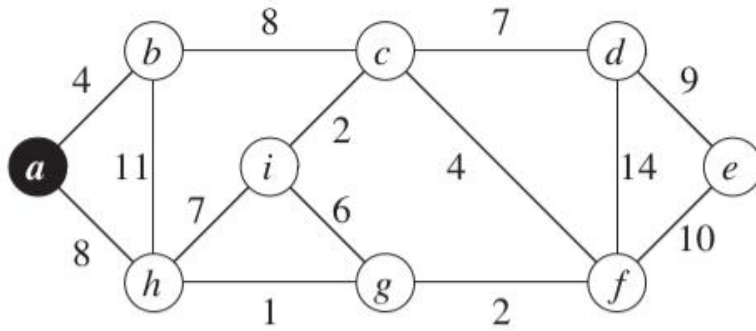
addEdge的方法比较容易理解。我们需要在Edge对象对应的两个节点里添加关联，所以有adj.get(v).add(e)和adj.get(w).add(e)。而对于edges方法来说，因为每个边有两个节点关联到上面，如果遍历所有节点的话，这些边会被遍历两次。为了避免重复的将边统计进来，我们用一个e.other() > v来设定一个条件，使得只有other方法返回值比当前v节点大的时候再统计。后面这部分的if(e.other(v) == v)情况是用来统计一些指向节点自身的边的情况。关于这个图的详细定义细节可以查看后续附件里的完整代码。

在定下来具体的图数据结构之后，我们来看看两个具体实现的算法。

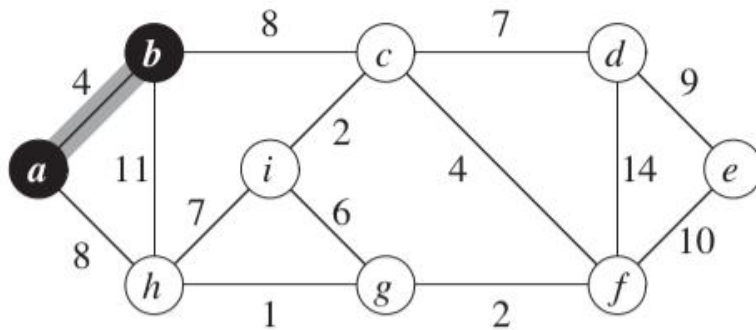
Prim算法

Prim算法的基本思路如下，首先选择任意一个节点作为一个单节点的树。它就相当于一棵树的根或者发起点。然后我们从这个节点开始，看它关联的所有边。每次我们选择一个边的时候，挑选一个权值最小的而且不在这个树的节点集合里的。因为如果我们增加一个边的时候，同时就把这个边所关联的另外一个节点加入到前面的树的节点集合里来了。

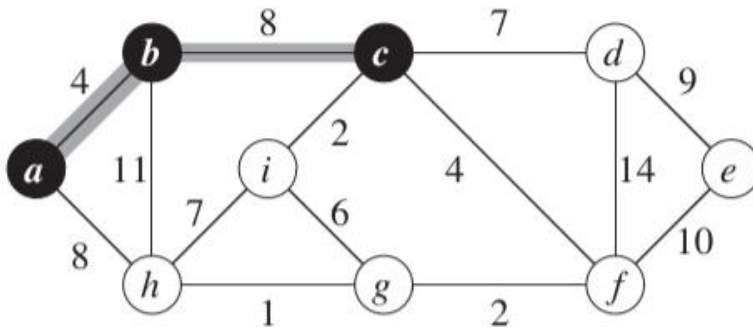
我们以以下的图来说明Prim算法的过程，首先，我们在图里选择节点a：



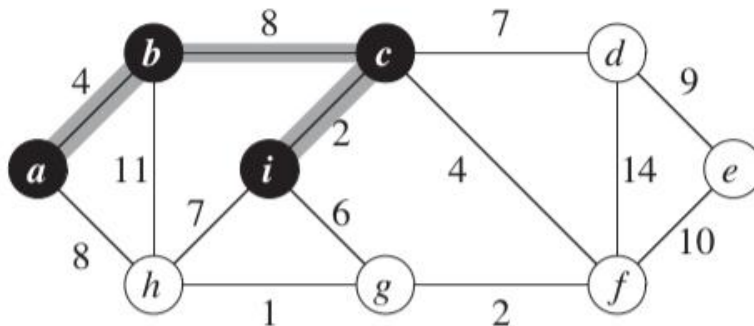
节点a是我们考虑的起始节点，按照算法的描述我们就要通过它来选择边，扩展整个树。a的边有两个分别关联到b和h，它们的权值分别为4和8。那么这时我们选择权值最小的边4, 这个边关联的节点b不在我们原来的树节点集合里。将这个边和关联的节点加入到树中间，这样我们树的集合里节点为{a, b}。如下图：



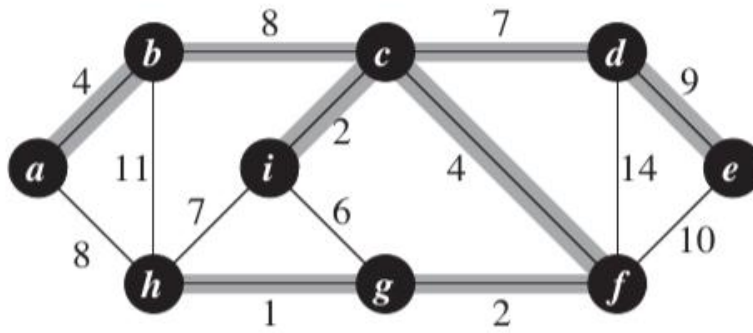
这个时候，我们要考虑和扩展的边就不仅仅是原来节点a的边了，也要考虑节点b的。从图中可以看到权值最小的边为bc, ah，它们的值都为8。因为c和h都不在树的节点集合里，所以它们都可以选取。假定我们选取bc。那么节点集合为{a, b, c}，其结构如下图：



按照前面同样的思路，c被加入到树里了，它关联的所有节点和边就需要和前面树里节点的边放到一起来考虑。所以这次我们选取到的节点是ci，它的权值为2, 而且i也不在节点集合里。加入后的节点集合为{a, b, c, i}，如下图：



按照前面讨论的过程，我们可以很容易推导出后面最终的树结构图：



总结起来，Prim算法无非就是首先找到一个节点，然后选择它关联的节点中权值最小的边，并将对应的节点也加入集合。然后将新加入的节点的边也加入到边选择考察的范围。这样重复前面的扩展过程，导致节点和边的队伍不断扩充壮大。

现在，从实现的角度来考虑，我们需要注意几个细节。一个就是，我们要考察的边必须放在某个地方保存起来，它们必然是我们的树节点集合里关联的边。这样每次我们能很方便的去选取它们最小的那个。另外一个就是，我们每次选择到一个边的时候还是需要判断这个新加入的点是否已经在树节点的集合里了，如果已经在就不能加这个边和节点。这两个问题，我们分别实现的思路如下。因为每次我们需要加入边，并且要选择最小的出来，我们不一定要对它们所有的进行排序，最有效的办法是采用一个最小堆。实际代码中可以使用PriorityQueue。而至于判断是否重复访问节点，我们可以定义一个和节点对应的boolean数组，每次访问到对应的节点时就将该数组里对应的元素值设置为true。

综合这些考虑，一个Prim算法的实现代码如下：

Java代码 ☆

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

```
public class LazyPrimMST {
    private double weight;
    private Queue<Edge> mst;
    private boolean[] marked;
    private Queue<Edge> pq;

    public LazyPrimMST(EdgeWeightedGraph g) {
        mst = new LinkedList<Edge>();
        pq = new PriorityQueue<Edge>();
        marked = new boolean[g.getVertices()];
        prim(g, 0);
    }

    private void prim(EdgeWeightedGraph g, int s) {
        visit(g, s);
        while(pq.size() > 0) {
            Edge e = pq.remove();
            int v = e.either(), w = e.other(v);
            if(marked[v] && marked[w]) continue;
            mst.add(e);
            weight += e.getWeight();
            if(!marked[v]) visit(g, v);
            if(!marked[w]) visit(g, w);
        }
    }

    private void visit(EdgeWeightedGraph g, int v) {
        marked[v] = true;
        for(Edge e : g.adj(v))
            if(!marked[e.other(v)])
                pq.add(e);
    }

    public Iterable<Edge> edges() {
        return mst;
    }

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

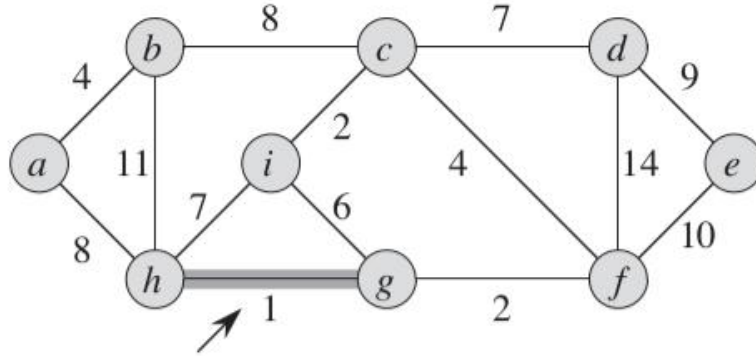
这里稍微截取了一部分代码。最关键的代码在prim()方法和visit()方法里。我们定义了pq来每次visit一个节点的时候将关联的边加入到其中。在加入之前我们只需要判断一下这个要访问的节点是否已经访问过。而prim方法里每次通过pq.remove()方法取出权值最小的边。这些选取出来的

边，至少有一个节点已经在树的节点集合里面了，所以我们之需要判断一下关联的节点里有一个不在，我们就可以去访问该节点。

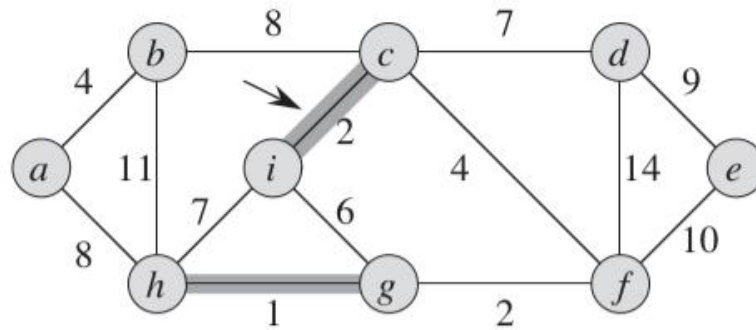
Kruskal算法

Kruskal算法考虑的思路和前面的不同。既然我们要找的Minimum Spanning Tree是要求涵盖所有节点并且权值最小。那么如果我们从权值最小的边这个角度入手呢？如果每次都选择权值最小的边，然后再考察它所关联的节点。假定我们图里的每个节点都是一个个独立的集合。它们每个集合就是包含它们本身。而一旦我们选择了一个边，相当于两个集合直接建立了一个关联，或者说将它们给合并了。比如最开始的时候，我们找到第一个边，那么它就将两个独立的节点合并成一个集合。然后我们再去权值最小的边。当然，并不是每次我们找到的权值最小的边就合适。比如说我们原来已经有几个节点在一棵连通的树里了，我们找到的边如果它的节点都在树的节点集合里就不合适。

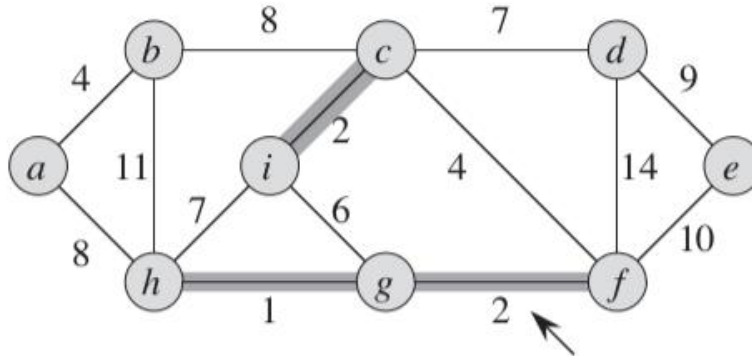
我们结合下面的图来详细的走一下后面的步骤。首先我们从图中权值最小的边，在这里是选择了hg，它的权值为1。



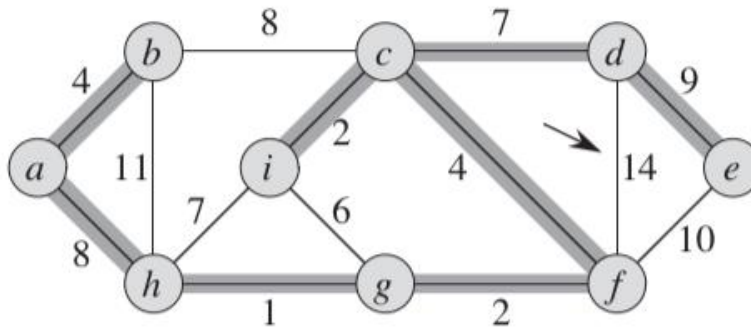
这里，我们选择的边将h,g关联起来，它们相当于形成了一棵树。然后，我们再选择下一个权值最小的边，这次我们找到了ci,gf，假定我们选择ci，则如下图：



在这里我们会发现一个有意思的地方。在不断引入最小权值边的时候，我们会引入一组组独立的集合，它们是相互关联的，但是暂时它们和其他的集合又是相互独立的。这时，我们再按照前面的思路挑最小的边，这次选择了gf：



在这里，我们可以反思一下。我们选择的边gf可以加入到hg的集合里。原因在哪里呢？无非就是因为这个权值为2的边gf一边g节点所涵盖的集合是{g,h}，而另外一边涵盖的节点集合是{f}，它们不一样，所以可以合并。所以，问题的最关键就在这里。我们每次选择权值最小的边，然后比较它两边关联的节点是否在同一个集合，如果不是则选取成功，否则需要继续选择后面的。按照这个思路，我们后面选取到的树结构如下：



现在，我们也从实现的角度来考虑一下细节。首先，我们需要将所有的边都放在一个地方好每次方便去权值最小的边。借用前面的思路，我们可以考虑最小堆。另外，每次找到的边就带来了新的节点。这些关联的节点就组成了一个集合。在不断导入边的过程中会形成多个集合。我们该用什么数据结构来描述它们呢？我们考虑到，每次我们需要加入元素引入集合。在引入新的边的时候又会带来集合之间的查找与合并。我们实际上可以使用Disjoint Set来处理。这个Disjoint Set是什么，有什么作用呢？我们这里先不详细介绍。我们暂时只需要知道它能够实现很好的集合合并操作，而且它查找一个元素是否在集合中的操作也非常有效。

对于我们前面讨论的Disjoint Set结构，它有两个比较常用的方法，一个是union()，主要是用来将两个元素归并到一个集合中。还有一个是connected，用来判断两个节点是否已经在同一个集合中。就是利用了这两个有效的方法，我们可以很容易的得出Kruskal算法的实现：

Java代码 ☆

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

```
public class KruskalMST {
    private Queue<Edge> mst;
    private double weight;

    public KruskalMST(EdgeWeightedGraph g) {
        mst = new LinkedList<Edge>();
        Queue<Edge> pq = new PriorityQueue<Edge>(g.edges());
        UF uf = new UF(g.getVertices());

        while(pq.size() > 0 && mst.size() < g.getVertices() - 1) {
            Edge e = pq.remove();
            int v = e.either(), w = e.other(v);
            if(uf.connected(v, w)) continue;
            uf.union(v, w);
            mst.add(e);
            weight += e.getWeight();
        }
    }

    public Iterable<Edge> edges() {
        return mst;
    }

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

在构造函数里，我们通过g.edges()将图里面所有的边取出来放到PriorityQueue里，然后不断的从里面取边。while循环的条件在于只要我们能够找到节点个数-1个边或者队列为空就可以了。所以循环里面的代码很简单，每次我们取出边，然后判断两边的节点是否属于同一个集合，是的话则忽略，不是的话则归并它们到同一个集合里。mst则用来保存所有选取出来的边。

这样，一个最小权值的树就这样构造出来了。前面我们提到的Disjoint Set到底是怎么整的，使得它们能够这么方便的处理集合呢？我们现在来看个究竟吧。

Disjoint Set


我们要考虑的这个集合和通常使用的Set还有点不一样。在jdk的类库里，Set的实现是基于一个HashMap来做的。在我们前面的问题场景里，一开始所有的节点相当于一个个独立的集合，相当于一开始有n个节点，就有n个集合。在后续通过引入边的时候将一些点合并。因此，问题的关键点就在于我们怎么有效的来表示和操作它们。

我们继续这么考虑，这里所有的节点相当于一个数组，对应的下标就是每个节点。所以，当我们引入一个边的时候，如果使得它们表示的值是相同的，以后其它并入这个集合的都设置成同样的值这不就可以表示同样的集合了吗？而如果我们比较两个节点是不是在同一个集合里，只要看它们对应的值是否相同就可以了。我们以一组数字组成的数组为示例来看：

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	8	5	6	7	8	9
6	5	0	1	2	8	8	5	6	7	8	9
		0	1	2	8	8	5	5	7	8	9
9	4	0	1	2	8	8	5	5	7	8	9
		0	1	2	8	8	5	5	7	8	8
2	1	0	1	2	8	8	5	5	7	8	8
		0	1	1	8	8	5	5	7	8	8

最开始的id[]列表里每个元素正好对应它们的下标。在加入p q = (4, 3)的时候，我们将4和3合并，也就是说将元素4的值修改成和3的一样。然后在加入(3, 8)的时候因为3和4已经是一个集合里的了，于是将3, 4的元素值都修改成和8一样。这样依次类推...

于是我们可以实现一个如下的集合定义：

Java代码 

```
public class UF {
    private int[] id;
    private int count;

    public UF(int n) {
        count = n;
        id = new int[n];
        for(int i = 0; i < n; i++)
            id[i] = i;
    }

    public int count() {
        return count;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public int find(int p) {
        return id[p];
    }

    public void union(int p, int q) {
        int pId = find(p);
        int qId = find(q);

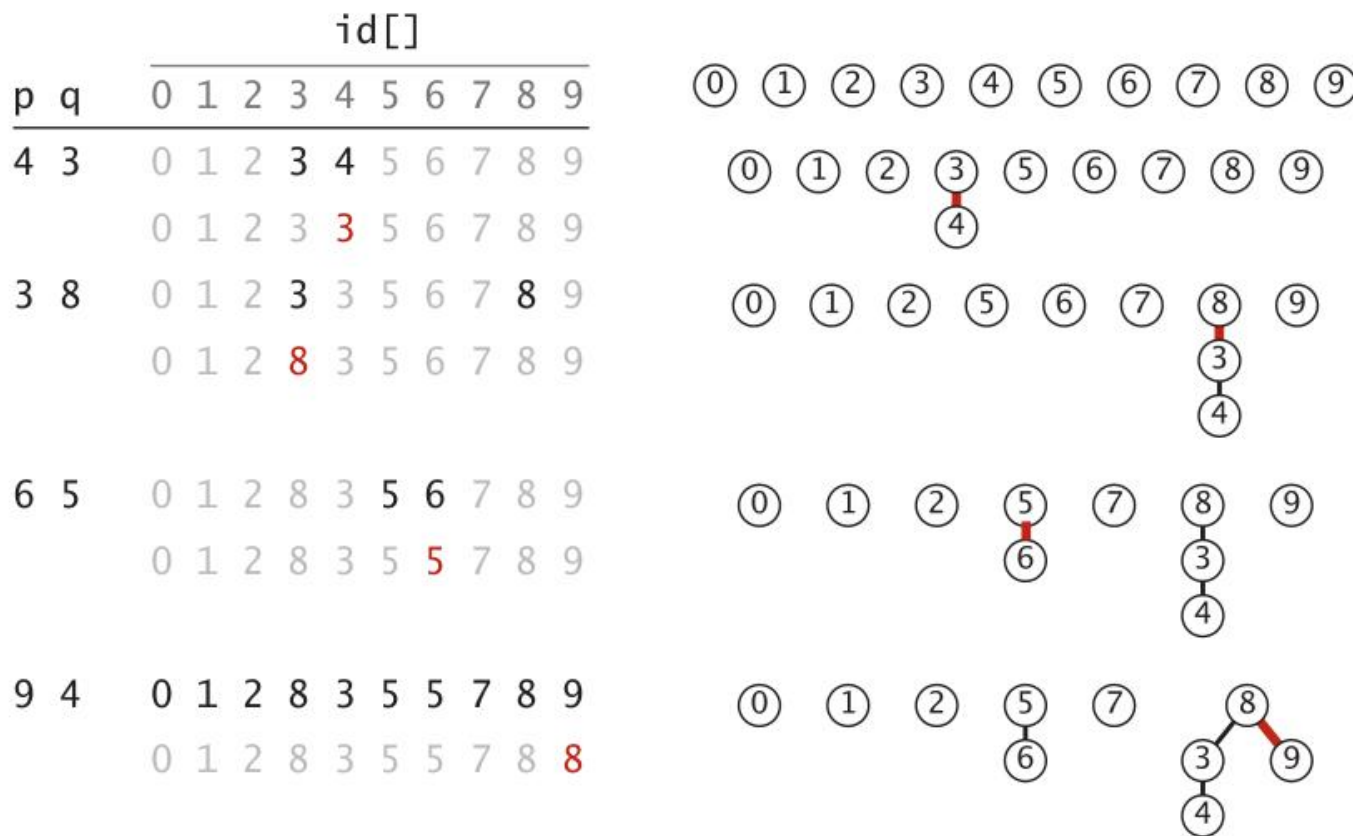
        if(pId == qId) return;

        for(int i = 0; i < id.length; i++)
            if(id[i] == pId) id[i] = qId;
        count--;
    }
}
```

我们这里定义的代码实现在于每次一个集合和另外一个新加入的元素合并时，就把集合里所有的元素对应值都统一成新加入元素的对应值。而我们知道，这里id[x]表示的就是元素x所对应的集合唯一标识。我们查找元素也很方便，只要返回id[x]就可以了。采用这种方式比较简单。不过每次集合合并的时候需要修改一个集合里所有的元素，在一些极端情况下，时间复杂度都可以到O(n)了。

前面的定义和方法能够实现很快的查找某个元素是否在某个集合里，因为find()方法只要访问一下数组的元素，它的时间消耗只是一个常量。只是在合并操作的时候会花的时间稍微多一点。除了前面的那种表示方法，其实我们还有一种方法，它使得集合合并的时间效率更高。

这种方法是基于这样一个思路的。前面我们每次要合并集合的时候都需要更新某个集合里所有的元素，可是如果我们换个角度来看。不管你某个集合怎么构造，它最开始合并的时候如果我们指定一个节点它作为另外一个节点的上一级，那么每次加入一个元素的时候，我们只需要合并两个集合的最上层，只要它们做一个调整就可以了，其它的元素只要设置为指向它自己的上级就可以了。这就好比两个公司合并，只要两个公司的高层指定一下结构和关系就行了，下面的人不需要管，该对谁负责的还是对谁负责。我们以下图为例说明一下整个过程：



在上图中，一开始每个节点是独立的。在加入(4, 3)的时候，我们设置4节点的上级为3，于是将它更新为3。然后再考虑(3, 8)，因为3节点所在的集合最高节点是3，而8节点就是本身，按照前面的原则，只要将3更新为8就可以了。(5, 6)也是如此。最有意思的就是(9, 4)，因为4节点所在的集合的最高节点需要递推。首先前面4的上一级节点是3，而3的上一级是8，所以9节点更新为8。所以说的更具体一点就是对于(a, b)中间两个节点，分别找到节点a和b的最高级节点，然后将a的最高级节点设置成b的最高级节点。于是，我们就有了另外一种实现：

Java代码

```
public class UF {
    private int[] id;
    private int count;

    public UF(int n) {
        if(n < 0)
            throw new IllegalArgumentException();
        count = n;
        id = new int[n];
        for(int i = 0; i < n; i++)
            id[i] = i;
    }

    public int count() {
        return count;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public int find(int p) {
        if(p < 0 || p > id.length)
            throw new IndexOutOfBoundsException();
        while(p != id[p])
            p = id[p];
        return p;
    }
}
```



```
public void union(int p, int q) {
    int pRoot = find(p);
    int qRoot = find(q);
    if(pRoot == qRoot)
        return;

    id[pRoot] = qRoot;

    count--;
}
```

在这里，如果我们需要找到某个节点所属于的集合，就相当于找它最高级节点，这个过程通过不断查找它的上一级节点，直到这个节点指向它自身就可以了。而union方法也很简单，更新一下一个节点集合最高节点的值就可以了。

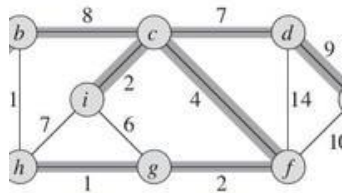
总结

Minimum Spanning Tree这个问题引出的两种典型解决方法的思路其实并不复杂。用几句话来概括一下的话，Prim算法是由点及边，扩散到整个图中。相当于一颗种子发芽的过程，不断向外扩展自己的根和枝叶。只是选择的这些节点都是相对自己来说代价最小的。而Kruskal算法的思路有点相反。它是以边带点。有点类似于农村包围城市的味道，每次它都是选择权值最小的边，将边对应的点加入到集合里来。通过一个最小权值的边来达到集合之间的互通有无。当然，它里面用到了disjoint set结构的特性以及最后构成树的边必然为节点个数-1的特性，使得它的实现也很简单直观。这两种算法的核心思想背后都是采用的贪婪算法的思想。

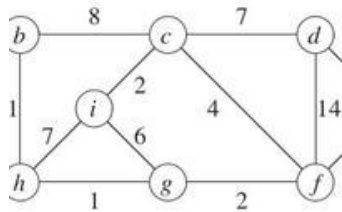
另外，里面运用到的disjoint set的结构在查找和归并一些集合数据的时候具有比较理想的性能，它和我们使用的Set类还不太一样。

参考材料

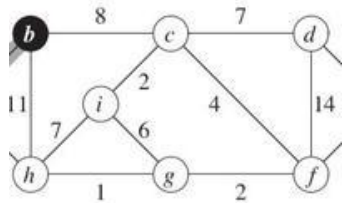
[Introduction to algorithms Algorithms](#)



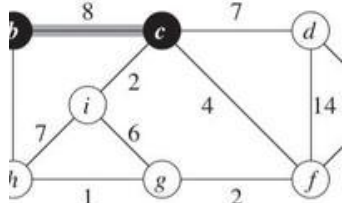
大小: 14.5 KB



大小: 12.7 KB



大小: 12.7 KB



大小: 12.7 KB