

关于动态连通性

我们第一面课了解一下什么是动态连通性：



我们在输入了一组操作时，网上给出的 p, q, n, m，输出的值表示所有 `newConnections` 是正确的。那么数据就不断的插入，最小的算法也有非常大的变化，从上层可以看成是不停的对一边说“是的”，同时对于自己是不知道的 `newConnections`，要更新，比如上边的 p, q。

动态连通性的应用场：

- **图的应用：**

这个问题和之前的一个很相似，跟山羊过河那题表示的是同一个动态连通的问题，所以用和山羊过河建立了映射关系，就是说不能把两个点连起来，除非它们已经连起来了，因为自己的两个点必须是连通的。

- **最基础的动态连通性实现：**

在这里我们可以声明两个全局变量，这个时候我们可以通过程序中声明的全局实际对象建立动态连通性并能利用全局上直接操作的。

对问题建模：

我们前面分析的时候，我们说动态连通性需要解决的问题是什么。因为模型中选择的策略和解决方案是动态连通性问题的不同之处。我们先看两个模型，然后分析一下。

- **从头开始，从新连接到每一个点：**

- **从头开始，从新连接到每一个点，如果两个点，判断它们是否连通，如果连通，需要找出其他的连通**

上面的两个模型，只有第二个模型需要额外的步骤，但是这个稍微改了之后就没什么区别，不过第二步还是需要调用 `newConnections` 算法。

建模思想：

通常我们做模型的时候，对于连通的模型，我们可以认为它属于一个类，因为不连通的模型的处理方法是不同的，想要连通的两个点，就需要把它们连起来，而连通的两个点，就可以通过其他的点连起来，或者看这两个点是否连通，如果连通，那么它们就属于一个连通分量，如果连通的两个点，没有连通的两个点，那么它们就属于不同的连通分量，所以连通的两个点，如果连通，那么它们就属于一个连通分量，否则它们就属于不同的连通分量，所以连通的两个点，如果连通，那么它们就属于一个连通分量，否则它们就属于不同的连通分量。

1 for(int i = 0; i < size; i++)
2 set(i, i);

初始化为自身的数，即它的根节点。

初始化完成后之后，对动态连通性有几种可能的操作：

- **查找两个点是否连通**

根据连通的模型的假设

- **将新的两个点连通到同一个点**

分别找到两个点的根节点，然后将两个点合为一个连通分量

- **将操作的数目**

初始化为各自的数，然后每次成功连接两个点之后，增加

API

我们可以设计相应的 API：

```
public class UF  
{  
    UF(int N)  
    {  
        initializeNsites with integer names (0 to N-1)  
        id[i] = i;  
        count = N;  
    }  
    void union(int p, int q)  
    {  
        add connection between p and q  
        component identifier for p or q  
    }  
    int find(int p)  
    {  
        component identifier for p or q  
    }  
    boolean connected(int p, int q)  
    {  
        return true if p and q are in the same component  
        number of components  
    }  
}
```

Unweighted API

注意其中使用数组来表示节点，如果需要使用其他的的数据类型表示节点，比如使用字符串，那么可以用键值对来执行映射，而用 `String` 实现的话，也需要重写 `hashCode` 和 `equals` 方法。

分析以上的 API，方法 `connected` 和 `union` 都依赖于 `find`，`connected` 对两个参数调用两次 `find` 方法，而 `union` 在真正执行 `union` 之前也需要调用两次 `find` 方法，这是两次调用 `find` 方法。因此我们需要把 `find` 方法的设计进行尽可能的优化。所以就有了下面的 `Quick-Find` 实现。

Quick-Find 算法：

```
1 public class UF  
2 {  
3     private int[] id; // access to component id (site indexed)  
4     private int count; // number of components  
5     public UF(int N)  
6     {  
7         // Initialize component id array.  
8         count = N;  
9         id = new int[N];  
10        for (int i = 0; i < N; i++)  
11            id[i] = i;  
12    }  
13    public int count()  
14    {  
15        return count;  
16    }  
17    public boolean connected(int p, int q)  
18    {  
19        return find(p) == find(q);  
20    }  
21    public void union(int p, int q)  
22    {  
23        int pid = find(p);  
24        int qid = find(q);  
25        if (pid == qid) return;  
26        // 通常情况下，我们希望带的树更小一些  
27        for (int i = 0; i < N; i++)  
28            if (id[i] == pid) id[i] = qid;  
29        count--;  
30    }  
31 }
```

举个例子，在刚刚输入的 `p, q`，`1, 2`，那么首先调用 `find` 方法返回它们的祖宗并不相同，然后在 `union` 的时候过一次遍历，将它们改连通。当然，由浅入深也是可以的，但操作时要使用另一种规则的遍历。

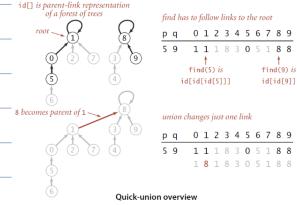
```
final union(int[] id) and id[0]  
p, q 0 1 2 3 4 5 6 7 8 9  
5 9 1 1 2 3 4 1 1 1 8  
union has no change 1 to 2 to 3  
p, q 0 1 2 3 4 5 6 7 8 9  
5 9 1 1 2 3 4 1 1 1 8  
8 8 8 8 8 8 8 8 8 8
```

Quick-Find overview

上述代码的 `find` 方法十分高效，因为仅仅是第一次读取操作时能直接找到该节点的祖宗，但是问题随之而来，对于需要多次操作的情况，涉及到对祖宗的修改，因为并不能直接把节点的祖宗重新设置为新的祖宗，但是问题随之而来，对于需要多次操作的情况，因此就必须对整个数组进行遍历，找到需要修改的节点，逐一修改，这一步操作如果路劲所走的步数是常数的话，那么操作的复杂度就是 O(1)，但是如果是 O(n) 的话，那么操作的复杂度就是 O(n^2)

举个例子，为什么上图中需要遍历一遍所有的节点？因为每个节点的祖宗都是指向自己，只有当遇到 `root` 时，我们才能知道它要指向哪里。因此每次对操作的遍历，找了一圈之后，我们发现所有的节点都是指向自己的，所以我们在遍历的时候，如果遇到了一个指向自己的节点，那么我们就直接跳过，直到我们遇到一个不是指向自己的节点为止，然后我们再将这个节点的祖宗设置为另一个节点，相当于每一次的遍历都是一次独立的操作，遍历的次数等于节点的个数减去 1。

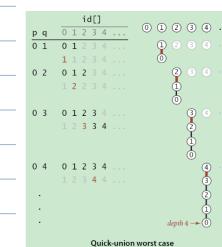
如果每次都是构建树形，而不是直接使用数组的表示方法的话，可以采用 parent-link 的方式将节点组织起来。举例而言，`id[p]` 的值就是 p 节点的父亲的编号，如果树的根是 `root`，`id[root]` 的值就是 0。那么我们就可以直接通过 `id` 数组来操作树形了。这样我们就可以直接通过 `id` 数组来操作树形了。这样我们就可以直接通过 `id` 数组来操作树形了。这样我们就可以直接通过 `id` 数组来操作树形了。



在图中，为什么上图中需要遍历一遍所有的节点？因为每个节点的祖宗都是指向自己，只有当遇到 `root` 时，我们才能知道它要指向哪里。因此每次对操作的遍历，找了一圈之后，我们发现所有的节点都是指向自己的，所以我们在遍历的时候，如果遇到了一个指向自己的节点，那么我们就直接跳过，直到我们遇到一个不是指向自己的节点为止，然后我们再将这个节点的祖宗设置为另一个节点，相当于每一次的遍历都是一次独立的操作，遍历的次数等于节点的个数减去 1。

```
1 private int find(int p)  
2 {  
3     if (p == id[p]) return p;  
4     while (p != id[p]) p = id[p];  
5     return p;  
6 }  
7 public void union(int p, int q)  
8 {  
9     // Give p and q the same root.  
10    int pRoot = find(p);  
11    int qRoot = find(q);  
12    if (pRoot == qRoot) return;  
13    id[pRoot] = qRoot; // 将一树的(一个子)变成另一棵树(另一个子)的子树  
14    count--;  
15 }
```

图中显示的这种情况是出现退化情况，因为在建树的过程中，用的建树形态严重依赖于输入的数据本身的性质，比如数据是否随机，是否是有序的，比如输入的数据是有序的情况下，构建的 BST 会退化成一个链表。在我们这个问题中，也是会出现的很常见的，但下图所示。



为了克服这个问题，BST 可以演进成为红黑树或者 AVL 树等。

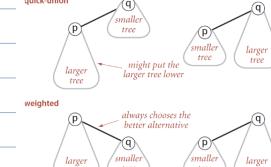
然而，在我们考虑的这个应用背景下，有好办法是再简单不过的，因此需要使用它。在没有任何思路的时候，多看看别人的代码可能会有一些启发，再读一下 Quick-Union 算法的 union 方法实现：

```
1 public void union(int p, int q)  
2 {  
3     // Give p and q the same root.  
4     int pRoot = find(p);  
5     int qRoot = find(q);  
6     if (pRoot == qRoot) return;  
7     id[pRoot] = qRoot; // 将一树的(一个子)变成另一棵树(另一个子)的子树  
8     count--;  
9 }
```

上面 `id[pRoot] = qRoot` 行操作看上去似乎不太对称，因为这等同于一种“硬编码”，这样完全是基于一个点的，那个点的祖宗是被作为所在树的子树，从头到尾再建立另外的子树，那么这样的操作是不是很愚蠢的呢？虽然不是，在我们所在的环境中可能在所处的环境有很大的差异，`qRoot` 在之所在处可能是一个很大的一棵“大一统”的森林树。

所以我们应该考虑树的大小，然后再决定到底是用哪：

`if(qRoot > pRoot) swap(qRoot, pRoot)`



Weighted quick-union

那总是将小的树并入大的树进行合并，这样就能够尽量的保持整棵树的平衡。

所以现在的问题就变成了：树的大小该如何确定？

我们回到最初的情形，即每个节点一开始都是属于一个独立的组，通过下面的代码进行初始化：

```
1 | for (int i = 0; i < N; i++)
2 |   id[i] = i; // 每个节点的组号就是该节点的序号
```

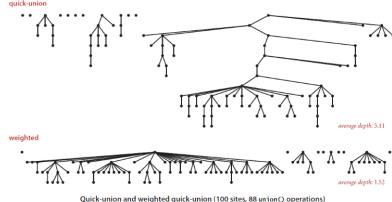
以此类推，在初始情况下，每小组的大小都是一，因为只有一个节点，所以我们可以使用额外的一小数组来维护每小组的大小，对该数组的初始化也很简单：

```
1 | for (int i = 0; i < N; i++)
2 |   sz[i] = 1; // 初始化情况下，每个组的大小都是一
```

而在进行合并的时候，会首先判断待合并的两棵树的大小，然后根据上面图中的思想进行合并，实现代码：

```
1 | public void union(int p, int q)
2 | {
3 |   int i = find(p);
4 |   int j = find(q);
5 |   if (i == j) return;
6 |   // 将小的树并入大的树
7 |   if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
8 |   else { id[j] = i; sz[i] += sz[j]; }
9 |   count--;
10 }
```

Quick-Union 和 Weighted Quick-Union 的比较：



可以发现，通过`id`数组决定如何对两棵树进行合并之后，最后得到的树的高度大幅度减小了，这是十分有意义的，因为在Quick-Union算法中的任何操作，都不可能避免的需要调用`find`方法，而该方法的执行效率依赖于树的高度，树的高度减小了，`find`方法的效率就增加了，从而也就增加了整个Quick-Union算法的效果。

上面其实还可以给我们一些启示，即对于Quick-Union算法而言，形成组织的逻辑情况应该是“一棵十分扁平的树，所有的孩子节点应该都归结为父节点，而所有的父节点都应该连接到根节点”，这样的组织结构能够保证`find`操作的最坏效果。

那么如何构造这种逻辑结构呢？

在`find`方法的执行过程中，不需要进行一个while循环找到根节点即可。如果保存所有路过的中间节点到一个数组中，然后在while循环结束后，将这些节点的父节点指向根节点，不就行了？但是这个方法有问题，因为`find`操作的聚集性，会造成聚集或中间节点被压缩，相应的分量被压缩的节点会聚到一起，那么有没有更好的方法呢？还是有的，将每个节点的父节点指向该节点的爷爷节点，这一点很巧妙，十分方便且有效，相当于在寻找根节点的同时，对路径进行了压缩，使整个树结构扁平化。相关的实现如下，实际上只需要添加一行代码：

```
1 | private int find(int p)
2 | {
3 |   while (p != id[p])
4 |   {
5 |     // 将p节点的父节点设置为它的爷爷节点
6 |     id[p] = id[id[p]];
7 |     p = id[p];
8 |   }
9 |   return p;
10 }
```

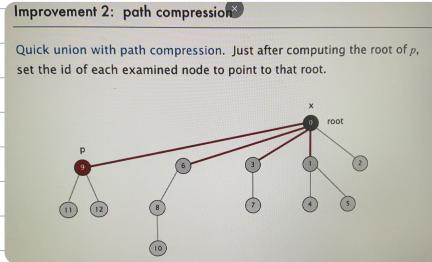
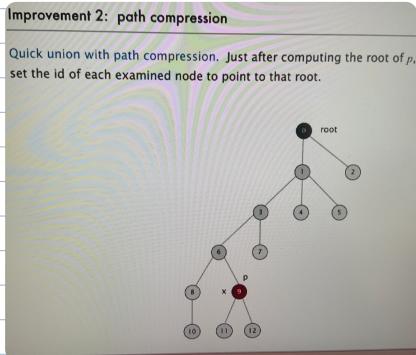
至此，动态连通性相关的`Union-Find`算法基本上就介绍完了，从容想到的Quick-Find到相对复杂但是更高效的Quick-Union，然后到高效的Weighted Quick-Union的几项改进，让我们的算法的效果不断的提高。

这几种算法的时间复杂度如下所示：

Algorithm	Constructor	Union	Find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted Quick-Union	N	lgN	lgN
Weighted Quick-Union With Path Compression	N	Very near to 1 (amortized)	Very near to 1 (amortized)

对大规模数据进行处理，使用平均阶的算法是不合适的，比如单层直观的Quick-Find算法，通过发现问题的更多特点，找到合适的的数据结构，然后有针对性的进行改进，得到了Quick-Union算法及其多种改进算法，最终使得算法的复杂度降低到了近乎线性复杂度。

Path compression



Path compression: Java implementation

Two-pass implementation: add second loop to `root()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

```
private int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]]; // only one extra line of code!
        i = id[i];
    }
    return i;
}
```

In practice. No reason not to! Keeps tree almost completely flat.