并查集算法的优化与实现

邓泉

摘要: 对并查集的优化算法进行调研，基于自己的思考和改进意见对这些算法进行整理分类，归纳了这些算法的特性，并用 C++代码实现，通过运行程序来比较了几种算法的性能。

关键词: 算法优化；并查集；加权；折叠

并查集（Union-find Set）也称 Disjoint Set，是集合的一种存储表示，结构简单，用途广泛，在数据结构的搜索及索引方法中应用较多，特别是在 等价类的算法处理中的应用尤显重要，用于处理一些不相交集合的合并及查询问题。

并查集能解决的问题一般可以转化为这样的形式：初始时 n 个元素各自分属不同的 n 个集合，通过不断的给出元素间的联系，要求实时的统计元素间的关系（即是否存在直接或间接的联系）。

下面讲述了并查集的基础实现以及分别对合并和查找操作的优化。

1 并查集基础实现

基础的并查集能实现以下三个操作（“并”“查”“集”三字由此而来）：

1. Union 合并两个集合；
2. Find 查找某个元素是否在一给定集合内（或查找一个元素所在的集合）；
3. UFsets （构造函数）建立一个集合。

1.1 quick_find 算法：

1.1.1 算法描述

quick_find 算法顾名思义，就是并查集查询操作快，而合并比较慢。按照并查集的定义进行数据结构和数据操作的声明，对 n 个不同元素划分成一组不相交的集合（即若干等价类），很自然地想到对每一个等价类（集合）编号，以连续的存储单元（即数组，data）来存储所有的个体元素，数组下标代表每一个单元素，而对应存储空间中的内容代表该个体元素所属集合的编号。

1.1.2 代码实现

```
int UFSets::Find(int x)
{
    return data[x];
}

void UFSets::Union(int a, int b)
{
    int Root1 = Find(a), Root2 = Find(b);
    if (Root1 == Root2) {
        return;
    }
    for (int i = 0; i < size; i++) {
        if (data[i] == Root2) {
            data[i] = Root1;
        }
    }
}
```

1.1.3 算法分析

实现方式采用（动态）数组，这个数组实际上反映了一颗高度最大为 2 的树。

对于 Find 查询操作，时间复杂度是 O(1)。而 Union 合并操作，由于需要遍历所有个体才能确保找到所有满足要求的个体元素，所以用到了所开辟的整个内存空间，每搜索一次的时间复杂度为 O(n)，这对于较大规模的问题处理起来是很不友好的，无疑需要对 Union 合并操作进行优化。

1.2 quick_union 算法：

1.2.1 算法描述

如果能通过一个元素找到同组的另外一个元素，这样便可以省去遍历所有元素的时间。通过树形结构来描述单个元素之间的关系，底层存储用父指针数组 parent[size]来存储，下标代表元素，采用树的父指针表示作为其存储表示，元素编号从 0 到 size - 1（size 是最大元素个数），那么第 i 个数组元素代表包含元素 i 的树结点，根结点的父指针为负数（为区分父指针信息），其绝对值表示集合的元素个数。

作者简介:

quick_union 算法的核心思想为，查询操作先判断两个不同元素所属树的根结点是否相同，若相同，则可以认为这两个元素是属于同一个集合；合并操作便是将其中一颗树的根结点的父指针指向另一颗树的根结点。

1.2.2 代码实现

```
int UFSets::Find(int x)
{
    while (parent[x] > 0) {
        x = parent[x];
    }
    return x;
}

void UFSets::Union(int a, int b)
{
    int Root1 = Find(a), Root2 = Find(b);
    parent[Root1] += parent[Root2];
    parent[Root2] = Root1;
};
```

1.2.3 算法分析

从上面的分析，合并和查找操作的时间复杂度为 O(H)，H 是树的高度，或者认为是 O(log n)。
相对 quick_find 实现的并查集，quick_union 牺牲了一点查找的性能，提高了合并的性能。

考虑到树的不平衡问题，最坏情况下，整棵树会退化成为一个单叉树（形如链表），这时时间复杂度就会变成 O(n)。例如将一棵高度为 n 的单叉树作为高度为 1 的树的子树，反复执行此操作，最后将会产生一棵退化的树。

1.3 quick_find 与 quick_union 的性能比较

1.3.1 测试一

元素个数 n = 100000，操作次数 count = 10000（Find 和 Union 各执行 10000 次），quick_find 与 quick_union 程序运行时间分别是 Time1，Time2。

运行结果：

Time1 = 0.610177652

Time2 = 0.009013917

不难看到，quick_union 的性能明显好于 quick_find。

1.3.2 测试二

n = 100000，count = 100000。

运行结果：

Time1 = 7.212751839

Time2 = 11.202537913

这种情况下，很可能是由于生成树的高度很大，而 quick_find 的 Find 操作的时间复杂度为 O(1)，quick_union 的 Find 和 Union 都是 O(H)，导致了 quick_find 比 quick_union 更快。

2 加权规则改进 Union 操作

2.1 基于 size（树结点个数）对 Union 的优化

2.1.1 算法描述

为了解决树的不平衡问题，在进行合并操作时，可首先判断两个集合的元素数量（两个树的结点数量），始终将结点数较少的一方作为另一方的子树，这样就能在一定程度上解决树的不平衡问题，从而提高 Find 和 Union 的效率。

例如：

初始时有如下的集合（图 1）：

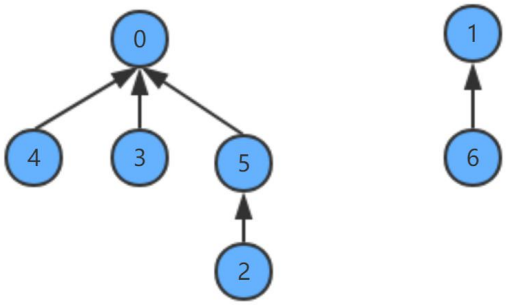

图 1

现在执行 Union(6, 2)，可以观察到有无加权规则的区别。

1. 执行 Union(6, 2)，没有加权规则（图 2）：

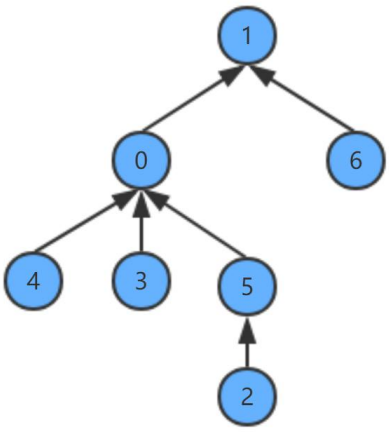

图 2

2. 执行 Union(6, 2)，基于加权规则（图 3）：

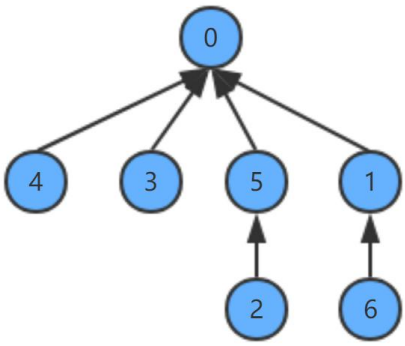

图 3

存储结构的设计上可以设一个新的 size 数组，用于表示每个元素所属集合的大小（树结点的个数）。（也可以不用，因为在 quick_union 算法中集合树根结点的指针表示已经反映了这个指标。）

2.1.2 代码实现

```
void UFSets::Union(int a, int b)
{
    int Root1 = Find(a), Root2 = Find(b), temp;
    if (size[Root1] != size[Root2]) {
        temp = parent[Root1] + parent[Root2];
        if (parent[Root2] < parent[Root1]) {
            parent[Root1] = Root2;
            parent[Root2] = temp;
        }
        else {
            parent[Root2] = Root1;
            parent[Root1] = temp;
        }
    }
};
```

2.2 基于 rank（树高度）对 Union 的优化

2.2.1 算法描述

上面基于 size 的优化方案，是结点数少的树往结点数多的树合并。显然存在一个问题，quick_union 算法的 Union 操作时间复杂度为 O(H)，取决于树的高度，结点数少不能等同于树的高度就小，那么在有些情况下基于 size 的优化方案费事不讨好。

例如：

初始时有如下的集合（图 4）：

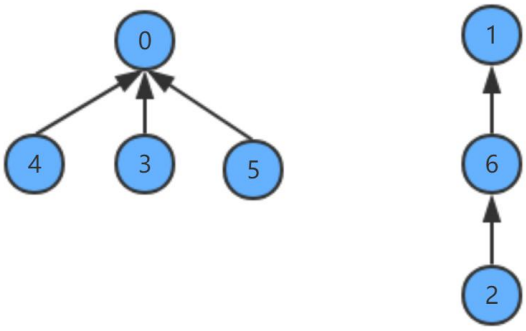

图 4

元素 3 所在的树总节点数有 4 个，但只有 2 层；元素 2 所在的树有 3 个节点，有 3 层。

执行 Union(3, 2)，比较以下两种情况：

1. 按照 size 的优化方案，执行 Union(3, 2)（图 5）：

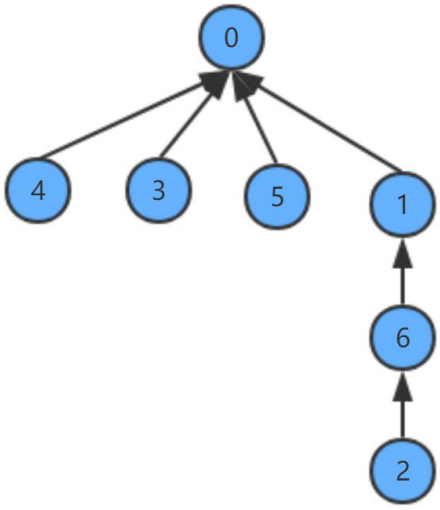

图 5

这个时候可以使用 rank 来优化（rank 代表数的高度或深度），核心思想为高度低的树向高度高的树合并。

2. 使用 rank 的优化方案，执行 Union(3, 2)（图 6）：

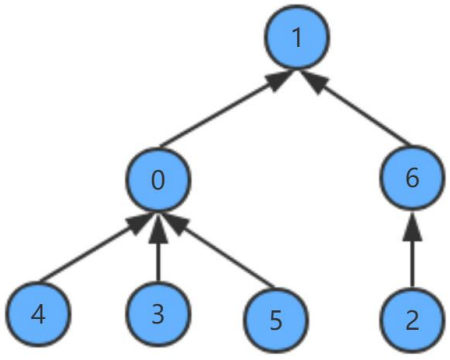

图 6

rank 意思是“秩”，表示高度的上界，实际上就是树的高度（或深度），这里认为只有一个节点的树的高度是 1，则其秩也是 1。基于 size 的优化方案，实质上就是在合并两棵树时，将高度较小的树合并到高度较大的树上。

两棵秩分别为 r1、r2 的树合并，如果秩不相等，我们将秩小的树合并到秩大的树上，这样就能保证新树的秩一定小于或等于原来的任意一棵树。如果 r1 与 r2 相等，两棵树任意合并，并令新树的秩为 r1 + 1。

存储结构的设计类似基于 size 的优化方案，可以设一个新的 rank 数组，用于表示每个元素所属集合的秩（树的高度）。

2.2.2 代码实现

```
void UFSets::Union(int a, int b)
{
    int Root1 = Find(a), Root2 = Find(b);
    if (rank[Root1] != rank[Root2]) {
        if (rank[Root1] < rank[Root2]) {
            parent[Root1] = Root2;
            rank[Root1] = rank[Root2];
        }
        else {
            parent[Root2] = Root1;
            rank[Root2] = rank[Root1];
        }
    }
    else {
        parent[Root2] = Root1;
        rank[Root1]++;
    }

};
```

2.3 性能分析

2.3.1 测试一

n = 100000, count = 100000

基于 size 与基于 rank 的优化算法程序运行时间分别是 Time3，Time4。

运行结果：

Time3 = 0.032306971

Time4 = 0.026279325

相同情况下：

Time1 = 7.212751839

Time2 = 11.202537913

在加权规则下的并查集优化方案，通过降低每棵树的高度，性能得到了极大的改善。

2.3.2 测试二

n = 10000000, count = 10000000

运行结果：

Time3 = 4.907501316

Time4 = 4.613372031

数据量非常大时，基于加权规则的并查集方案带来的优化影响更显著。

3 折叠规则改进 Find 操作

加权规则能够通过改进 Union 操作，降低了 Find 的时间复杂度。但随着问题规模增大，结点数增多，多次合并以后树高仍然会相当大，这时进一步想，理想情况下最好能建立一个 n 叉树（图 7），那么 Find 查找操作的时间复杂度就仅仅是常数级别的 O（1）了。

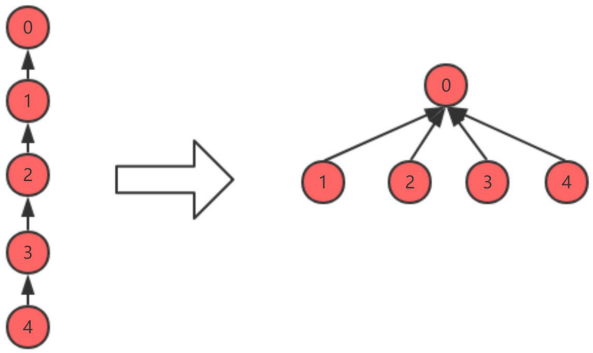

图 7

对于一个集合树来说，它的根节点下面可以依附着许多结点，尝试在查找的过程中自底向上，如果此时访问的节点不是根节点的话，那么我们可以把这个节点尽量的往上挪一挪，这样就可以有效减少了树的高度，这个树由高变矮的过程用一个形象的词来形容，那就是折叠。

3.1 路径压缩

3.1.1 算法描述

在 quick_union 算法执行的 Find 操作中：从一个结点，不断的通过 parent 父指针数组向上去寻找他的父结点，直至找到祖先结点。在这个过程中，我们相当于走过了从这个结点到根结点的这条路径，其上的所有结点也就给遍历了一遍，时间复杂度为 O(H)，H 为树的高度。

采取这样的折叠策略：在寻找的过程中，沿着轨迹不断将途中的结点连接到祖先节点上（即将指针指向祖先节点）。

例如：

初始时有如下的并查集（图 8），

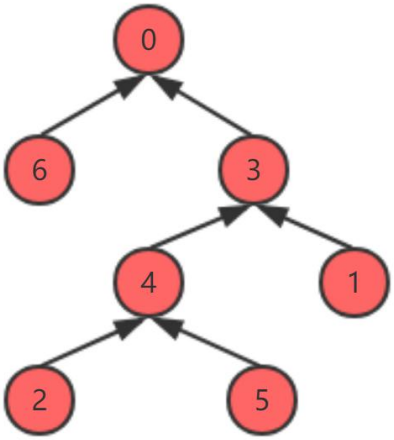

图 8

执行操作 Find(2)。

Step 1：从 2 压缩路径。结点 2 指向根结点 0。（图 9）

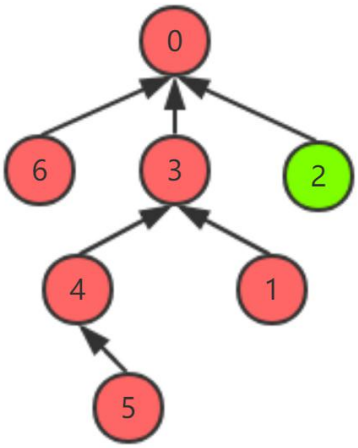

图 9

Step 2：从 4 压缩路径。结点 4 指向根结点 0，树的层数由原来的 4 层变成了 3 层，高度减一。（图 10）

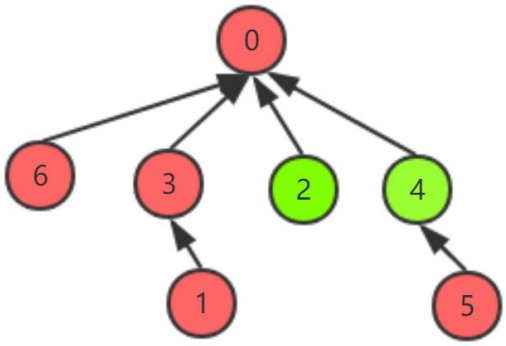

图 11

Step 3：从 3 压缩路径。结点 3 原来的父指针就指向根结点 0，这一步集合树没有发生改变。（图 12）

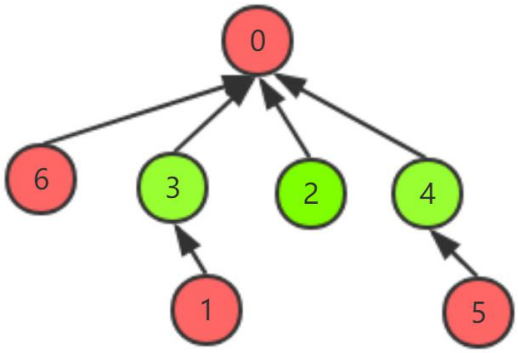

图 12

通过上面的过程可以看到，在进行 Find 查找操作的同时，不仅把需要查找的根节点给找到了，与此同时还对树进行了压缩，树的高度变得尽可能地少了，这样在下一次执行 Find 操作的时候，那么效率将会大大的提高。

可以预想到，执行很多次这样的压缩操作后，集合树很可能出现只有两层的最理想情况，此时所有的非根结点都直接指向根节点，这种情况下搜索都只需要一步就能够完成。

3.1.2 代码实现

有递归与迭代两种实现方式。

```
//递归
int UFSets::Find(int x)
{
    if (parent[x] >= 0) {
        parent[x] = Find(parent[x]);    //将途中结点（包括 x 自己）连接到根结点上
    }
    return x;
}

//非递归
int UFSets::Find(int x)
{
    int rt = x;                         //先找到根结点 rt
    while (parent[rt] >= 0) {
        rt = parent[rt];
    }
    while (rt != x) {
        int temp = parent[x];
        parent[x] = rt;
        x = temp;
    }
    return rt;

    }
```

3.2 路径分裂

3.2.1 算法描述

核心思想与路径压缩相似，不同之处在于：在 Find 的过程中，使路径上的每个结点都指向其祖父结点，即父结点的父结点。

（为防止发生越界问题，这里对 parent 数组做一个小修改，那就是根结点的父指针表示不再等于集合元素数量的相反数了，而是指向自身。这样的话，如果该元素的父亲节点正好是根节点，那么让其指向父亲节点的父亲节点并不会出错，根据根元素的父亲节点指向其自己的结构，此时父亲节点的父亲节点仍然是有效的，即还是根节点。）

例如：

起始时的并查集如下图所示（图 13），

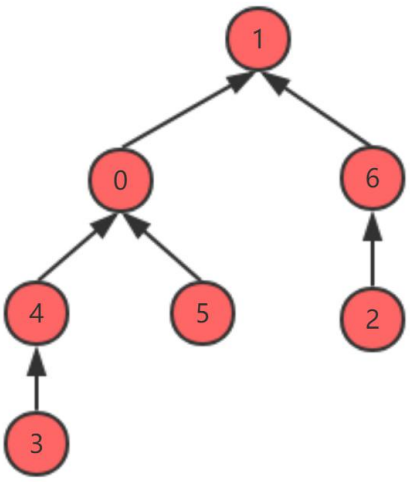

图 13

执行操作 Find(3)。

Step 1：从 3 分裂路径。结点 3 指向祖父结点 0，树的层数由原来的 5 层变成了 4 层，高度减一。（图 14）

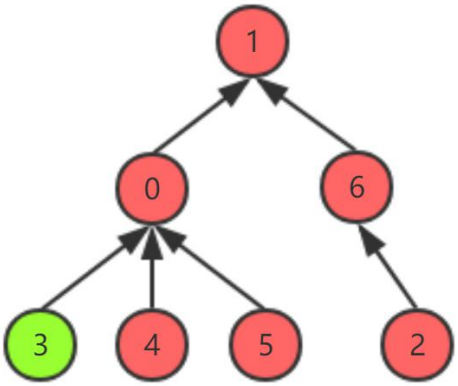

图 14

Step 2：从 4 分裂路径。结点 4 指向祖父结点 1。（图 15）

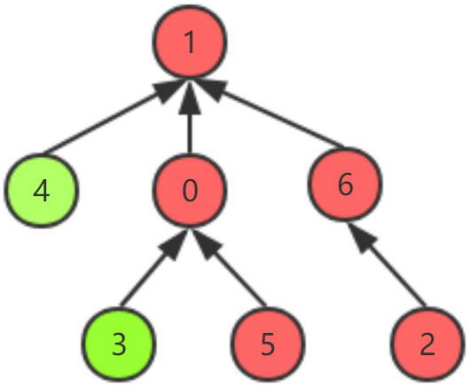

图 15

Step 3：从 0 分裂路径。结点 0 的父结点即为根结点，实际上没做实质性的操作。（图 16）

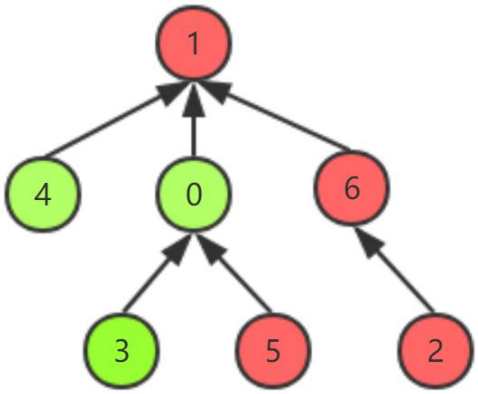

图 16

3.2.2 代码实现

```
int UFSets::Find(int x)
{
    while (x != parent[x]) {
        int temp = parent[x];
//保存 x 的父节点
        parent[x] = parent[parent[x]];
//将 x 指向祖父节点
        x = temp;
//更新 x 为其原来的父结点
    }
    return x;
```

```
}
```

3.3 路径减半

3.3.1 算法描述

路径分裂和的优化的考虑在于：如果让每一个结点都指向根结点那么时间开销较大，不妨选择折中方案，指向离自己较近的结点（祖父结点），这样既能让树的高度不至于过大，查找操作的时间复杂度也会更好。-路径减半算法在此基础上更进一步：在 Find 的过程中，使路径上每隔一个节点就指向它的祖父节点。这里的 parent 数组设计与路径分裂算法中的相同。

例如：

起始时的并查集如下图所示（图 17），

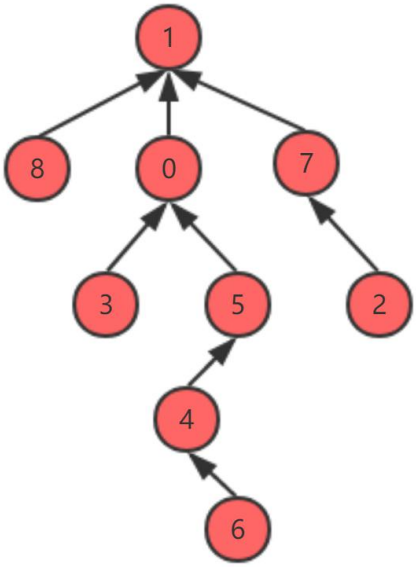

图 17

执行操作 Find(6)。

Step 1：从 6 分裂路径。结点 6 指向祖父结点 5，树的层数由原来的 5 层变成了 4 层，高度减一。（图 18）

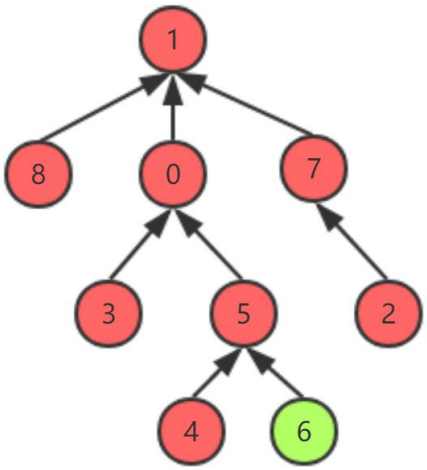

图 18

Step 2：从 5 分裂路径。结点 5 指向祖父结点 1，树的层数由原来的 4 层变成了 3 层，高度减一。（图 19）

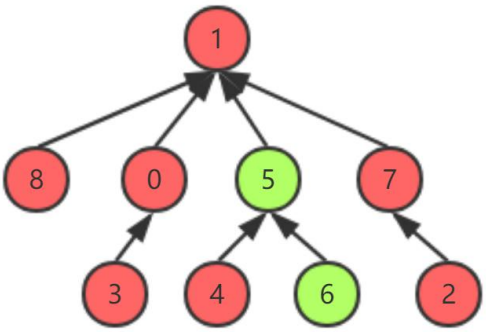

图 19

Step 3：从 1 分裂路径。结点 1 即为根结点，树没有变动。（图 20）

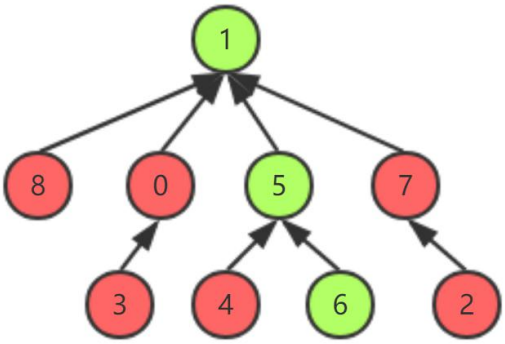

图 20

3.3.2 代码实现

```
int UFSets::Find(int x)
{
    while (x != parent[x]) {
        parent[x] = parent[parent[x]];
//将 x 指向祖父节点
        x = parent[x];
//更新 x 为刚指向的祖父结点
    }
    return x;
}
```

3.4 性能分析

3.3.1 测试

n = 10000000, count = 10000000

路径压缩算法的程序运行时间是 Time5。

运行结果：

Time5 = 4.395936423

相同情况下：

Time3 = 4.907501316

Time4 = 4.613372031

3.4.2 理论分析

资料显示，使用折叠规则结合加权规则的优化可以保证每种操作的均摊时间达到 O(α(n))，这样是最理想的，这里的α(n) 是 Ackermann 函数的反函数，值小于 5，所以并查集操作本质上需要花费常数级的时间。

4 结语

至此，并查集的优化与实现就完成了，整体的优化思路，可以概括为：

quick_find --> quick_union --> 加权（size, rank） --> 折叠（路径压缩，路径分裂，路径减半）

对于并查集可以用来处理哪一类的问题、并查集是怎么一步步进行优化的，还要不断进一步体会其中的精髓。对于不同的应用场景，并查集会有不同的变种形式，应在上述优化思想的基础上灵活运用，结合多种优化并借助实验验证才能得到最佳的优化方案。

参考文献：

[1] Robert Sedgewick,Kevin Wayne 著,谢路云 译.算法（第4版）[M].北京:人民邮电出版社,2012 年 10 月第一版