并查集 Disjoint Set Union

庄博伟

ZUCC ACM Group

December 7th 2021

目录

从一个问题开始

初始时 n 个人，每个人分属一个家族，接下来执行 m 次操作（合并 or 询问）：

- 合并操作会选择两个人 x, y，表示将 x, y 所在的 2 个家族合并成 1 个家族（若本来就属于同一个家族则不合并）
- 询问操作会选择两个人 x, y，问是否属于同一家族
- $n \leq 10^5, m \leq 10^5$

暴力解

对每个家族维护一个 vector，并对每个人维护一个所属家族

```
void init(int n) {
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
        vec[i].push_back(i);
    }
}

bool query(int x, int y) {
    return fa[x] == fa[y];
}

void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

时间复杂度

init 和 query 的时间复杂度好算，分别为 $O(n)$ 和 $O(1)$，那么 merge 呢?

```
void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

时间复杂度

init 和 query 的时间复杂度好算，分别为 $O(n)$ 和 $O(1)$，那么 merge 呢?

```
void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

一次 merge 的时间复杂度应为 $O(|vec_{fy}|)$

时间复杂度

init 和 query 的时间复杂度好算，分别为 $O(n)$ 和 $O(1)$，那么 merge 呢?

```
void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

一次 merge 的时间复杂度应为 $O(|vec_{fy}|)$

最坏情况

假设一共进行 $n-1$ 次合并，第 i 次合并 i 与 $i+1$，相当于每次将前 i 个人都迁移到 $i+1$ 的家族中。

则会进行 $1+2+3+\cdots+n-1$ 次操作，故复杂度为 $O(n^2)$ 

目录

1. 从一个问题开始
2. 神奇的启发式合并
3. 并查集

优化 merge

让我们加一个很容易想到，但看起来很不靠谱的优化

```
void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    // if (vec[fx].size() < vec[fy].size()) swap(fx, fy);
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

优化 merge

让我们加一个很容易想到，但看起来很不靠谱的优化

```cpp
void merge(int x, int y) {
    int fx = fa[x];
    int fy = fa[y];
    // if (vec[fx].size() < vec[fy].size()) swap(fx, fy);
    if (fx != fy) {
        for (int i : vec[fy]) {
            fa[i] = fx;
            vec[fx].push_back(i);
        }
        vec[fy].clear();
    }
}
```

时间复杂度

merge 的最坏复杂度将一口气降为 $O(n \log n)$！

时间复杂度证明

- 每次合并时，都是小集合并入大集合，那么对于小集合来说，其每次合并，所处的家族大小至少变成原先的两倍

时间复杂度证明

- 每次合并时，都是小集合并入大集合，那么对于小集合来说，其每次合并，所处的家族大小至少变成原先的两倍
- 对于开始时候的单个元素（单人家族），他每次处在 fy（被合并的一方）时，所处的家族大小至少变成原先的两倍

时间复杂度证明

- 每次合并时，都是小集合并入大集合，那么对于小集合来说，其每次合并，所处的家族大小至少变成原先的两倍
- 对于开始时候的单个元素（单人家族），他每次处在 fy（被合并的一方）时，所处的家族大小至少变成原先的两倍
- 推论：对于每个人，最多作为 fy 的一员被合并 $log_2 n$ 次

时间复杂度证明

- 每次合并时，都是小集合并入大集合，那么对于小集合来说，其每次合并，所处的家族大小至少变成原先的两倍
- 对于开始时候的单个元素（单人家族），他每次处在 fy（被合并的一方）时，所处的家族大小至少变成原先的两倍
- 推论：对于每个人，最多作为 fy 的一员被合并 $log_2 n$ 次
- 换句话说所有元素都最多在 fy 中出现 $log_2 n$ 次，因此 $\sum |vec[fy]| \leq nlog_2 n$

目录

另一种优化方法

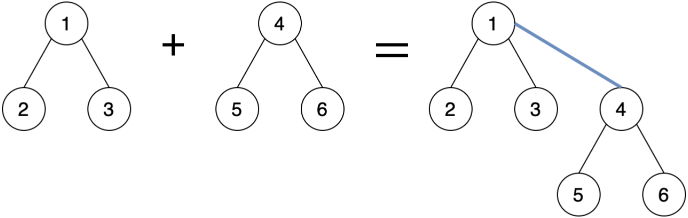
- 用树形结构存储一个家族，即给每个人安排一位长辈（除了根节点，即整个家族的祖先）

另一种优化方法

- 用树形结构存储一个家族，即给每个人安排一位长辈（除了根节点，即整个家族的祖先）
- 判断两个人是否属于同一家族，等价于判断两人的祖先是否是同一人

另一种优化方法

- 用树形结构存储一个家族，即给每个人安排一位长辈（除了根节点，即整个家族的祖先）
- 判断两个人是否属于同一家族，等价于判断两人的祖先是否是同一人
- 而 merge 操作则转化为两颗树合并，将一颗树的根变为另一颗树的根即可



Code

find(x) 表示寻找 x 的祖先，即树根，sz[x] 则用来存家族（树）的大小

```
void init(int n) {
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
        sz[i] = 1;
    }
}

int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}

bool query(int x, int y) {
    return find(x) == find(y);
}

void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx != fy) {
        fa[fy] = fx;
        sz[fx] += sz[fy];
        sz[fy] = 0;
    }
}
```

时间复杂度

- init 的复杂度还是 $O(n)$
- query 和 merge 的复杂度基本在 2 次 find 上，因此 find 的复杂度基本可以视作整个代码的复杂度

时间复杂度

- init 的复杂度还是 $O(n)$
- query 和 merge 的复杂度基本在 2 次 find 上，因此 find 的复杂度基本可以视作整个代码的复杂度
- 而单次 find(x) 的复杂度和 x 离树根的距离（结点的深度）有关，也就是 $O(depth_x)$

```
int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}
```

时间复杂度

- init 的复杂度还是 $O(n)$
- query 和 merge 的复杂度基本在 2 次 find 上，因此 find 的复杂度基本可以视作整个代码的复杂度
- 而单次 find(x) 的复杂度和 x 离树根的距离（结点的深度）有关，也就是 $O(depth_x)$

```
int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}
```

那么最坏时间复杂度是多少呢?

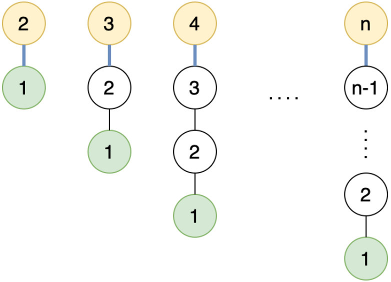
最坏复杂度

我们可以用和之前类似的方法构造出最坏的情况

最坏复杂度

我们可以用和之前类似的方法构造出最坏的情况

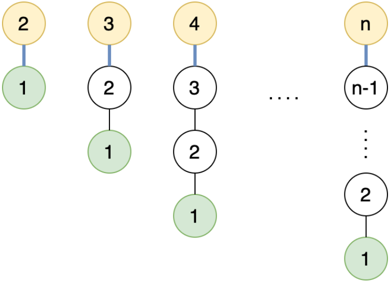
- 进行 $n-1$ 次合并，第 i 次合并$i+1$和1，即 1 所在的集合作为 fy
- 合并的过程相当于一条链不断伸长的过程



最坏复杂度

我们可以用和之前类似的方法构造出最坏的情况

- 进行 $n-1$ 次合并，第 i 次合并 $i+1$ 和 1，即 1 所在的集合作为 fy
- 合并的过程相当于一条链不断伸长的过程



最坏复杂度

find(1) 的复杂度会随着合并的进行而线性增长，共执行

$1+2+3\cdots+n-1$ 次，因此最坏时间复杂度为 $O(n^2)$ 

按秩合并（启发式合并）

使用和启发式合并类似的方法优化 merge

```
void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    // if (sz[fx] < sz[fy]) swap(fx, fy);
    if (fx != fy) {
        fa[fy] = fx;
        sz[fx] += sz[fy];
        sz[fy] = 0;
    }
}
```

按秩合并（启发式合并）

使用和启发式合并类似的方法优化 merge

```
void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    // if (sz[fx] < sz[fy]) swap(fx, fy);
    if (fx != fy) {
        fa[fy] = fx;
        sz[fx] += sz[fy];
        sz[fy] = 0;
    }
}
```

- 每个点所在的树，倘若该点深度增加，说明他作为 fy（被合并的一方）被合并了一次，整棵树的大小至少变为原先的 2 倍

按秩合并（启发式合并）

使用和启发式合并类似的方法优化 merge

```
void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    // if (sz[fx] < sz[fy]) swap(fx, fy);
    if (fx != fy) {
        fa[fy] = fx;
        sz[fx] += sz[fy];
        sz[fy] = 0;
    }
}
```

- 每个点所在的树，倘若该点深度增加，说明他作为 fy（被合并的一方）被合并了一次，整棵树的大小至少变为原先的 2 倍
- 因此不会有树高（树高 = 树上最深的点的深度）超过 $log_2 n$ 的树，所有点的深度均在 $log_2 n$ 以内

按秩合并（启发式合并）

使用和启发式合并类似的方法优化 merge

```
void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    // if (sz[fx] < sz[fy]) swap(fx, fy);
    if (fx != fy) {
        fa[fy] = fx;
        sz[fx] += sz[fy];
        sz[fy] = 0;
    }
}
```

- 每个点所在的树，倘若该点深度增加，说明他作为 fy（被合并的一方）被合并了一次，整棵树的大小至少变为原先的 2 倍
- 因此不会有树高（树高 = 树上最深的点的深度）超过 $log_2 n$ 的树，所有点的深度均在 $log_2 n$ 以内

时间复杂度

因此单次 $find(x)$ 的复杂度均在 $O(log_2 n)$ 以内，整体复杂度为

$O(mlog_2 n)$ 

路径压缩

我们来重新看一下之前的 find

```
int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}
```

路径压缩

我们来重新看一下之前的 find

```
int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}
```

- 其实在找到 x 的祖先 fx 后，可以让 x 直接变为 fx 的亲儿子
- 毕竟我们只需要保证每个点的祖先是正确合理的，就可以处理询问，而保证祖先正确的情况下自然是离祖先越近越好

路径压缩

我们来重新看一下之前的 find

```
int find(int x) {
    while (fa[x] != x) {
        x = fa[x];
    }
    return x;
}
```

- 其实在找到 x 的祖先 fx 后，可以让 x 直接变为 fx 的亲儿子
- 毕竟我们只需要保证每个点的祖先是正确合理的，就可以处理询问，而保证祖先正确的情况下自然是离祖先越近越好
- 在 x 寻找祖先路上碰到的所有点都可以变成 fx 的亲儿子！

```
int find(int x) {
    int rt = x;
    while (fa[rt] != rt) {
        rt = fa[rt];
    }
    while (x != rt) {
        int tmp = fa[x];
        fa[x] = rt;
        x = tmp;
    }
    return rt;
}
```

时间复杂度

不使用按秩合并，仅使用路径压缩的情况下，最坏复杂度同样为 $O(m\log_2 n)$
但是其证明需要一种叫二项树的东西，因此此处摸了（

摸　了

想看的可以自己看看这篇博客

优雅的递归！

通过递归，我们可以写出一些非常优雅的路径压缩代码！

优雅的递归！

通过递归，我们可以写出一些非常优雅的路径压缩代码！

```
int find(int x) {
    if (x == fa[x]) {
        return x;
    } else {
        return fa[x] = find(fa[x]);
    }
}
```

小知识：赋值操作也是有返回值的

优雅的递归！

通过递归，我们可以写出一些非常优雅的路径压缩代码！

```
int find(int x) {
    if (x == fa[x]) {
        return x;
    } else {
        return fa[x] = find(fa[x]);
    }
}
```

小知识：赋值操作也是有返回值的

再使用三目运算符压一下行～

优雅的递归！

通过递归，我们可以写出一些非常优雅的路径压缩代码！

```
int find(int x) {
    if (x == fa[x]) {
        return x;
    } else {
        return fa[x] = find(fa[x]);
    }
}
```

小知识：赋值操作也是有返回值的

再使用三目运算符压一下行～

```
int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
```

ohhhhhh，一个只有一行的路径压缩并查集就写好了！

时间复杂度

按秩合并和路径压缩这两个优化并不冲突，可以一起用！

时间复杂度

按秩合并和路径压缩这两个优化并不冲突，可以一起用！

优化	最坏时间复杂度
没有优化	$O(n^2)$
按秩合并	$O(m\log_2 n)$
路径压缩	$O(m\log_2 n)$
按秩合并 + 路径压缩	$O(m\alpha(n))$

其中 α 为阿克曼函数的反函数，可以当成一个不会超过 5 的常数，更详细的可以自己看维基百科

其他资料

听不懂的同学可以看看其他地方讲的，总有一份能听懂的：

- OI-wiki
- 知乎
- BIT 寒假集训讲课录播，这里还有些别的好玩的也可以看看。

END

END