并查集

dbywsc

2025/8

目录

并查集(union-find disjoint sets) 是一种用于管理元素所属集合的数据结构。本质上，并查集是一个森林
并查集支持以下操作:
合并 (Union)：合并两个元素的所属集合
查询 (Find)：查询某个元素的所属集合

P3367 【模板】并查集

要想实现并查集，首先要进行**初始化**操作。
要想维护并查集，本质上维护的是对于每一个节点，它的父节点是谁。而在初始状态下，节点之间没有合并关系，因此我们认为它们一开始的父节点是它们自己。关于维护父节点是谁，我们可以用数组 $p[1..n]$ 来存储。

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

想要查询节点的根节点，需要不断地向当前节点的父节点移动，直到到达根节点。

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

然而，这样操作需要不断的递归，时间复杂度很高。因此我们可以使用路径压缩操作。

对于在同一个联通块中的节点，我们发现它们之间无论是怎样连接的，最终都连向同一个根节点。因此在查询时，我们可以顺便维护一下节点的连接关系，如果查询的节点刚好属于这个联通块，我们直接让它连向根节点。

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

接下来考虑如何将两个联通块合并。
合并的逻辑其实非常简单，对于两个不同的联通块，我们可以让任意一个向另一个合并，此时直接让一个联通块的根节点连向另一个点根节点即可[1]。

```
void unite(int x, int y) {
    int px = find(x), int py = find(y);
    if(px == py) return;
    p[px] = py;
}
```

[1]由于我们在分别查询两个联通块的根节点时，顺带对两个联通块都做了路径压缩操作，所以合并的时间复杂度也不会非常高。

除了维护节点之间的连接关系，我们还可以维护并查集中每个联通块中点的数量。

与维护连接关系类似，维护大小我们可以使用一个数字 $siz[1..n]$ 来完成。

最开始时，每个节点自己是一个联通块，因此它们的根节点是它们自己；它们的大小为 1。

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

可以发现，是否维护 $siz[1..n]$ 都不影响查询操作。[2]
而在合并操作时，我们只需要在改变归属关系之后，加一下两个联通块的大小就能维护 siz 。但是，在维护了大小后，我们可以进一步优化合并操作。
显然，对于两棵深度、节点数都不一的树，选择不同的树的根节点进行连接对后续的时间复杂度的影响是不一样的。为了防止引起时间复杂度退化，我们可以每次选择节点更少或者深度更小的树，让它连向另一棵树。

```
void unite(int x, int y) {
    int px = find(x), py = find(y);
    if(px == py) return;
    if(siz[px] < siz[py]) std::swap(x, y);
    siz[px] += siz[py];
    p[py] = px;
}
```

[2]同学们可能会发现，如果使用了路径压缩，那么对于某些节点，查询后它们的 siz 理应发生改变。事实上，维护大小时，我们只需要保证根节点的大小是正确的即可。

上面的操作我们叫做**启发式合并**。
关于并查集的时间复杂度，要想证明比较困难，因此我们直接给出结论：在只使用了路径压缩而不使用启发式合并的情况下，并查集的最坏时间复杂度为 $O(m \log n)$ ，平均为 $O(m\alpha(n))$。
而在同时使用了启发式合并和路径压缩后，并查集的平均时间复杂度**均摊**为 $O(\alpha(n))$。
显然，并查集的空间复杂度为 $O(n)$ 。

在本页中，我们将介绍 $\alpha(n)$ 是什么，有兴趣的同学可以课后查阅。

$\alpha(n)$ 是 $A(m,n)$ ，即 *Ackermann* 函数的反函数。

Ackermann 函数的定义为：

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

而 $\alpha(n)$ 是 *Ackermmann* 的反函数，即为最大的整数 m 使得 $A(m,m)\leq n$ 。

P1551 亲戚

在本题中，我们可以把亲戚关系看作一个联通块或者一张树形图。那么如果两个人有亲戚关系，就将他们合并，而判断两人是否有这样的亲戚关系，只需要查询一下是否在一个联通块中就好了。

```cpp
void init(void) {
    for(int i = 1; i <= n; i++) p[i] = i, siz[i] = 1;
}
int find(int x) {
    if(x == p[x]) return p[x];
    return p[x] = find(p[x]);
}
void unite(int x, int y) {
    int px = find(x), py = find(y);
    if(px == py) return;
    if(siz[px] < siz[py]) std::swap(px, py);
    siz[px] += siz[py];
    p[py] = px;
}
void solve(void) {
    std::cin >> n >> m >> P; init();
    for(int i = 1; i <= m; i++) {
        int x, y; std::cin >> x >> y; unite(x, y);
    }
    for(int i = 1; i <= P; i++) {
        int x, y; std::cin >> x >> y;
        std::cout << (find(x) == find(y) ? "Yes" : "No") << endl;
    }
}
```

维护并查集时，我们可以在并查集的边上定义某种权值，以及这种权值在路径压缩时产生的运算，从而解决更多的问题。

P2024 [NOI2001] 食物链

首先考虑一定为假话的情况：

当 $x > n$ 或者 $y > n$ 时是假话；

当 x 吃 x 时是假话。这两种我们可以直接忽略，对于之后的操作，就需要判断是否与之前的关系冲突。

我们发现，当 a 吃 b 、b 吃 c 时，一定有 c 吃 a，因此，我们可以建立一个并查集，同时维护两个节点之间的距离。

形如 $a \to b$ 的操作，我们定义 a 和 b 之间的距离 d 为 1，如果又出现了 $b \to c$ 的操作，那么 b 到 c 之间到距离为 1，此时 a 到 c 之间到距离为 2，即 $a \leftarrow c$ 。

如果出现了 $a \to b$，$b \leftarrow c$ 的情况，那么 a 与 c 同类，此时 a 和 c 的距离为 3 。

对于上述的所有操作，我们可以在路径压缩时，将子节点到父节点之间的距离转换成子节点到根节点之间的距离。由于只有三种操作，此时子节点到根节点的距离只会出现 $\mod 3$ 后分别为 $0, 1, 2$ 的情况，分别对应等价，吃，被吃三种关系。
因此我们在维护距离后，每当出现新的判断时，直接观察新判断后出现的距离是否与之前的距离冲突就可以了。

```
int find(int x) {
    if(x != p[x]) {
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }
    return p[x];
}
```

```cpp
void solve(void) {
    std::cin >> n >> m;init();
    while(m--) {
        int t, x, y; std::cin >> t >> x >> y;
        if(x > n || y > n) {ans++; continue;}
        int px = find(x), py = find(y);
        if(t == 1) {
            if(px == py && (d[x] - d[y]) % 3) ans++;
            else if(px != py) {
                p[px] = py;
                d[px] = d[y] - d[x];
            }
        } else {
            if(px == py && (d[x] - d[y] - 1) % 3) ans++;
            else if(px != py) {
                p[px] = py;
                d[px] = d[y] + 1 - d[x];
            }
        }
    }
    std::cout << ans << endl;
}
```

并查集本身是一个非常实用的数据结构，并且有许多图论算法也是在并查集的基础上实现的，因此它的使用非常广泛。
除了我们本节介绍的带权并查集之外，并查集还有许多别的扩展应用，大家有兴趣的话可以自行了解。
P3958 [NOIP 2017 提高组] 奶酪
P1111 修复公路
P1536 村村通
P1892 [BalticOI 2003] 团伙
P1991 无线通讯网