算法入门-并查集

Junlist

2025 年 11 月 26 日

1 什么是并查集?

并查集（Disjoint-Set Union）是一种用于处理不相交集合合并与查询的数据结构，它支持两种核心操作：查找（确定元素属于哪个集合）和合并（将两个集合合并为一个）。通过维护一棵逻辑树结构，其中每个节点指向其父节点。在实际的算法实现时，我们常常用数组来模拟这种逻辑上的树。

2 两种并查集

2.1 第一种并查集

- 用编号最小的元素标记所在集合；
- 定义一个数组 Set[1..n]，其中Set[i] 表示元素i所在的集合；

Set(i)	1	2	1	4	2	6	1	6	2	2
i	1	2	3	4	5	6	7	8	9	10

不相交集合：$\{1,3,7\}$, $\{4\}$, $\{2,5,9,10\}$, $\{6,8\}$
这种方式下的查找与合并的函数分别是：

```
int find(int x)
{
    return Set[x];
}
```

Listing 1: find

```
void merge(int a, int b)
{
    int i = min(Set[a], Set[b]);
    int j = max(Set[a], Set[b]);

    for(int k = 0; k < size; ++k)
    {
        if(Set[k] == j)
        {
            Set[k] = i;
        }
    }
}
```

Listing 2: merge

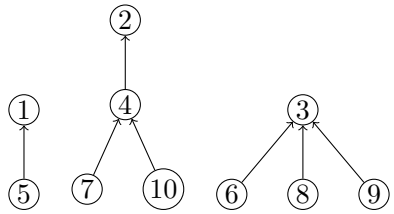
我们很容易知道在以这种方式实现的并查集 $find$ 这个函数的时间复杂度为 $O(1)$,而 $merge$ 函数无论并查集的逻辑形式如何其时间复杂度统一为 $O(n)$。这是我们无法再继续优化的，当面对大量数据的合并工作时可能会超时。

2.2 第二种并查集

每个集合用一棵”有根树”表示，定义数组 Set[1..n]

- Set[i] = i，则i表示本集合，若是集合对应树的根
- Set[i] = j，若j不等于i，则j是i的父节点

Set(i)	1	2	3	2	1	3	4	3	3	4
i	1	2	3	4	5	6	7	8	9	10



这种方式下的查找与合并的函数分别是：

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

Listing 3: find

```
void merge(int a, int b)
{
    Set[a] = b;
}
```

Listing 4: merge

在这种情况下我们不难发现 $merge$ 函数的时间复杂度为 $O(1)$ ，而此时的 $find$ 操作的空间复杂度的最差情况是当 Set 的逻辑结构为链状树的时候其时间复杂度为 $O(n)$。但是这种情况下我们有优化的空间，我们可以在查找或者合并的过程中改变树的结构让其下一次查找变得更加迅速：

2.2.1 按秩合并(不常用)

方法：将深度小的树合并到深度大的树

实现：假设两棵树的深度分别为h_1和h_2，则合并后的树的高度h是：

$$h = \begin{cases} \max(h_1, h_2), & \text{若 } h_1 \neq h_2 \\ h_1 + 1, & \text{若 } h_1 = h_2 \end{cases}$$

效果：任意顺序的合并操作以后，包含k个节点的树的最大高度不超过$\lfloor \log k \rfloor$ 。

这种优化下的查找与合并的函数分别是：

```
int find(int x)
{
    while(Set[x] != x)
    {
```

```
        x = Set[x];
    }
    return x;
}
```

Listing 5: find

```
void merge(int a, int b)
{
    if (Height[a] == Height[b])
    {
        Height[a] += 1;
        Set[b] = a;
    }
    else if (Height[a] < Height[b])
    {
        Set[a] = b;
    }
    else
    {
        Set[b] = a;
    }
    Height[a] = Height[b] = max(Height[a], Height[b]);
}
```

Listing 6: merge

经过我们对 $merge$ 逻辑的修改我们可以在不改变整体树的逻辑意义而将 $find$ 的时间复杂度由 $O(n)$ 降低至 $O(\log n)$。但是由于这种逻辑下我们需要维护更多的信息而且其真正意义上的实现十分不便所以我们不常用。

2.2.2 路径压缩

- **解决思想：**每次查找的时候，如果路径较长，则修改信息，以便下次查找的时候速度更快
- **具体方案：**第一步，找到根结点。第二步，修改查找路径上的所有节点，将它们都指向根结点

这种优化下的查找与合并的函数分别是：

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

Listing 7: find

```
void merge(int a, int b)
{
    int fa = find(a);
    int fb = find(b);

    if(fa != fb)
    {
        Set[fa] = fb;
    }
}
```

Listing 8: merge

在这种优化下我们边查找某一个节点的祖先，查找这个节点的祖先的路径中的节点均会在经历一次查找后均指向其祖先节点，在经历多次合并和查找后可认为二者的时间复杂度均为 $O(1)$。

3 KursKal最小生成树算法

Kruskal算法是一种用于在加权连通图中寻找最小生成树的算法。该算法的目标是找出图中边的子集，使得这些边构成的树包含图中的所有顶点，并且树的总权值（边的权值之和）最小。

算法步骤：

1. 将图中的所有边按权值从小到大排序。
2. 初始化一个空的最小生成树，开始时最小生成树不包含任何边。

3. 遍历排序后的边，对于每条边，如果它连接的两个顶点属于不同的连通分量，则将它加入最小生成树中，并合并这两个顶点所在的连通分量。

4. 重复步骤3，直到最小生成树中包含所有顶点或者已经选择了$n-1$条边（n是顶点的数量）。

Algorithm 1 Kruskal算法：求解最小生成树

```
procedure KRUSKAL(G(V, E))
    将边集 E 按权重从小到大排序
    初始化并查集，每个顶点自成一个集合
    MST ← ∅                                          ▷ 最小生成树的边集
    edgesAdded ← 0                                   ▷ 已添加的边数
    for 每条边 (u, v, w) 按权重从小到大顺序 do
        if edgesAdded = |V| − 1 then
            break                                    ▷ 已找到最小生成树
        end if
        if Find(u) ≠ Find(v) then                    ▷ u和v不在同一连通分量
            Union(u, v)                              ▷ 合并两个连通分量
            MST ← MST ∪ {(u, v, w)}
            edgesAdded ← edgesAdded + 1
        end if
    end for
    return MST
end procedure
```