基本的并查集（Disjoint Set Union）数据结构

黄京

Jul 25, 2025

并查集（Disjoint Set Union, DSU）是一种高效处理不相交集合合并与查询操作的数据结构。其核心作用在于管理元素分组，支持动态连通性问题的解决，例如判断两个元素是否属于同一集合或合并不同集合。这种数据结构之所以重要，是因为它在处理大规模数据集时提供近乎常数时间的均摊性能，远优于传统方法。典型应用场景包括无向图的连通分量检测（如识别网络中的孤立组件）、Kruskal 最小生成树算法中的边筛选、游戏开发中的像素区域连通性分析，以及编译器优化中的变量等价性判断。这些场景都需要频繁执行集合操作，而并查集通过其独特设计实现了高效支持。

1　核心概念与术语

并查集的核心结构基于森林表示法，其中每个集合被建模为一棵树，树根节点作为集合的代表元（Root），唯一标识整个集合。关键操作包括「find」和「union」：find(x) 用于查找元素 x 所属集合的代表元，而 union(x, y) 则合并元素 x 和 y 所在的集合。存储方式通常采用父指针数组（如 parent[i]），其中每个元素 i 存储其父节点的索引。这种设计允许高效地追踪集合关系，但需要优化以避免性能退化。

2　基础实现与问题分析

基础实现有两种常见方式。第一种是 Quick-Find 实现，通过数组直接存储集合 ID；查找操作时间复杂度为 O(1)，但合并操作需要更新所有相关元素的 ID，导致 O(n) 时间，效率低下。第二种是 Quick-Union 实现，采用树形结构存储父指针；合并操作仅需修改父指针（O(1)），但查找操作在最坏情况下可能退化至 O(n)，当树结构退化为链表时性能大幅下降。关键问题在于如何避免这种退化，确保树保持平衡，从而提升整体效率。

3　优化策略

优化并查集的核心策略包括按秩合并（Union by Rank）和路径压缩（Path Compression）。按秩合并的思路是在合并操作中让深度较小的树依附于深度较大的树，避免树高无谓增长；实现时维护一个 rank 数组记录树高的上界，时间复杂度分析表明这能限制树高增长。路径压缩则在 find 操作中扁平化路径，直接将所有节点指向根节点；可通过递归或迭代方式实现，例如递归版本中在查找过程中更新父指针。组合使用这两种优化后，均摊时间复杂度降至 O(α(n))，其中 α 是反阿克曼函数，对实际数据 α(n) 通常小于 5，近乎常数时间。

4 完整代码实现（Python 示例）

以下是使用路径压缩和按秩合并的 Python 实现：

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n)) # 初始化父指针数组，每个节点自成一集合
        self.rank = [0] * n # 初始化秩数组，记录树高的上界

    def find(self, x):
        if self.parent[x] != x: # 如果当前节点不是根节点
            self.parent[x] = self.find(self.parent[x]) # 递归压缩路径，更新父指针指向根节点
        return self.parent[x] # 返回根节点作为代表元

    def union(self, x, y):
        rx, ry = self.find(x), self.find(y) # 查找两个元素的根节点
        if rx == ry: # 如果根节点相同，表示元素已在同一集合
            return False # 合并失败
        if self.rank[rx] < self.rank[ry]: # 按秩合并：如果 rx 的秩较小
            self.parent[rx] = ry # 让 rx 依附于 ry
        elif self.rank[rx] > self.rank[ry]: # 如果 ry 的秩较小
            self.parent[ry] = rx # 让 ry 依附于 rx
        else: # 秩相等时
            self.parent[ry] = rx # 任意选择 rx 为父节点
            self.rank[rx] += 1 # 秩增加 1，确保树高不增长
        return True # 合并成功
```

代码解读：初始化阶段 `__init__` 设置每个节点为独立集合（`parent[i] = i`）和初始秩为 0。`find` 方法使用递归实现路径压缩，当节点非根时递归调用并更新父指针，最终返回根节点。`union` 方法先调用 `find` 获取根节点，若不同则按秩合并；比较 `rank` 值决定依附关系，秩相等时增加父节点的秩以防止树退化。这种实现确保操作高效。

5 复杂度分析

并查集的空间复杂度为 $O(n)$，主要来自存储父指针和秩数组。时间复杂度方面，单次操作（`find` 或 `union`）的均摊成本为 $O(\alpha(n))$，其中 $\alpha(n)$ 是增长极慢的反阿克曼函数，实际应用中通常小于 5。相比之下，未优化版本（如 Quick-Union）的最坏时间复杂度可达 $O(n)$，优化后性能提升显著，尤其在大规模数据下优势明显。

6 实战应用案例

实战中并查集广泛用于连通性问题。案例一：计算无向图的连通分量；步骤包括初始化 DSU、遍历所有边执行 union 操作合并连通节点，最后统计唯一根节点数量即为分量数。案例二：Kruskal 最小生成树算法；并查集高效判断边是否连接不同分量，伪代码片段如遍历排序边并调用 union，仅当返回 True 时添加边至生成树。案例三：LeetCode 547 朋友圈问题；给定关系矩阵，使用 DSU 合并直接或间接朋友，最终根节点数即为朋友圈数量。

7 常见问题与陷阱

常见问题包括秩（rank）与实际高度的区别；秩是理论高度的上界，代码中更易维护，无需在路径压缩后更新。路径压缩不影响秩，因为秩仅用于合并决策。统计集合数量时，需遍历所有节点并计数唯一根节点。动态增加节点需扩展父数组（例如使用哈希表或可扩展数组），但基础实现不支持删除操作，需特殊设计。

8 扩展变种

扩展变种包括带权并查集，维护节点与根的额外关系如距离或奇偶性（用于问题如等式约束）。动态连通性支持删除操作涉及更复杂策略，如懒删除或代理节点。并行并查集针对分布式环境优化，例如分片处理或异步合并。并查集的核心价值在于高效处理动态集合合并与查询，优化后均摊时间复杂度近乎 O(1)。其适用场景集中于连通性问题、图算法和动态等价关系管理。学习建议强调理解优化原理（路径压缩和按秩组合）而非机械记忆代码，这有助于应对变种问题。掌握并查集能显著提升算法设计能力。