深入理解并查集（Disjoint Set Union）

叶家炜

Jul 15, 2025

在计算机科学中，动态连通性问题是一个经典挑战。想象一个社交网络场景：用户 A 和 B 成为好友后，我们需要快速判断任意两个用户是否属于同一个朋友圈。传统方法如深度优先搜索（DFS）或广度优先搜索（BFS）能处理静态图，但当关系动态变化时（如频繁添加或删除好友），这些方法效率低下。每次查询都需要 O(n) 时间重建连通性，无法应对大规模数据。并查集（Disjoint Set Union）应运而生，它支持近常数时间的合并（union）与查询（find）操作，时间复杂度为 O(α(n))，其中 α(n) 是反阿克曼函数，增长极其缓慢；空间复杂度仅为 O(n)。本文将深入剖析并查集的核心原理，手把手实现两种关键优化（路径压缩和按秩合并），并通过实战代码解决算法问题。

1　并查集核心概念剖析

并查集的逻辑结构基于森林表示法：每个集合用一棵树表示，树根作为代表元（代表该集合）。初始时，每个元素自成集合；合并操作将两棵树连接，查询操作通过查找根节点判断元素所属集合。例如，元素 1、2、3 初始为独立集合，合并 1 和 2 后，它们共享同一个根。存储结构使用 `parent[]` 数组：`parent[i]` 存储元素 i 的父节点索引。初始化时，每个元素是自身的根，即 `parent[i] = i`。核心操作包括 `find(x)`（查找 x 的根）和 `union(x, y)`（合并 x 和 y 所在集合），这些操作确保了高效的动态处理能力。

2　暴力实现与性能痛点

基础版并查集未引入优化，代码简单但性能存在瓶颈。以下是 Python 实现：

```
class NaiveDSU:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, x):
        while self.parent[x] != x: # 暴力爬树：沿父节点链向上遍历
            x = self.parent[x]
        return x

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
```

```
        if rootX != rootY:
            self.parent[rootY] = rootX # 任意合并：可能导致树高度暴涨
```

在 find 方法中，通过 while 循环向上遍历父节点链，直到找到根节点。union 方法先调用 find 定位根节点，再将一个根指向另一个。问题在于：合并时若任意将小树挂到大树下，树可能退化成链表。例如，连续合并形成链式结构后，find 操作需遍历所有节点，时间复杂度恶化至 O(n)，无法处理大规模操作（如 10^6 次查询）。

3 优化策略一：路径压缩（Path Compression）

路径压缩的核心思想是在查询过程中扁平化访问路径，减少后续查询深度。具体分为两步变种：隔代压缩（在遍历时跳过父节点）和彻底压缩（递归压扁整个路径）。彻底压缩版效率更高，代码实现如下：

```
def find(self, x):
    if self.parent[x] != x:
        self.parent[x] = self.find(self.parent[x]) # 递归调用：将当前节点父指针直接指向根
    return self.parent[x]
```

在 find 方法中，递归调用 self.find(self.parent[x]) 不仅返回根节点，还将 x 的父指针直接更新为根。例如，若路径为 x → p → root，递归后 x 和 p 都指向 root。这使树高度大幅降低，单次查询均摊时间复杂度优化至 O(α (n))，显著提升吞吐量。实际测试中，10^6 次查询耗时从秒级降至毫秒级。

4 优化策略二：按秩合并（Union by Rank）

按秩合并通过控制树高度增长避免退化。秩（Rank）定义为树高度的上界（非精确高度），合并时总是将小树挂到大树下。代码增强如下：

```
class OptimizedDSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n # 秩数组：初始高度为 0

    def union(self, x, y):
        rootX, rootY = self.find(x), self.find(y)
        if rootX == rootY: return

        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY # 小树根指向大树根
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else: # 高度相同时
            self.parent[rootY] = rootX
            self.rank[rootX] += 1 # 更新秩：高度增加
```

在 union 方法中，比较根节点秩大小：若 rank[rootX] < rank[rootY]，则将 rootX 挂到 rootY 下；高度相同时，任意合并并将新根的秩加 1。这确保树高度增长受控（最坏情况 O(log n)），避免链式结构。例如，合并两个高度为 2 的树时，新树高度为 3，而非暴力实现的随意增长。

5 复杂度分析：反阿克曼函数之谜

优化后（路径压缩 + 按秩合并），并查集操作的时间复杂度为 O(α(n))。α(n) 是反阿克曼函数，定义为阿克曼函数 A(n, n) 的反函数，增长极缓慢：在宇宙原子数（约 10^80）范围内，α(n) < 5。数学上，阿克曼函数递归定义为：

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

α(n) 是满足 A(k, k) ⩾ n 的最小 k 值，其缓慢增长特性使并查集在工程中视为近常数时间。性能对比实验显示：10^6 次操作下，未优化版耗时 >1000ms，优化版仅需 <50ms，差异显著。

6 实战应用场景

并查集在算法竞赛和工程中广泛应用。经典算法题如 LeetCode 547 朋友圈问题：给定 n×n 矩阵表示好友关系，求朋友圈数量。解法中初始化并查集，遍历矩阵，若 M[i][j] = 1 则调用 union(i, j)，最后统计根节点数量。另一个场景是检测无向图环：遍历每条边，若 find(u) == find(v) 则存在环；否则调用 union(u, v)。这作为 Kruskal 最小生成树算法的前置步骤：排序边权重后，用并查集合并安全边。工程中，游戏地图动态计算连通区域（如玩家移动后更新区块连接），或编译器分析变量等价类（如类型推导），都依赖并查集的高效动态处理。

7 完整代码实现（Python 版）

以下是结合路径压缩和按秩合并的优化版并查集：

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n)) # 父指针数组：初始化每个元素自成一集合
        self.rank = [0] * n # 秩数组：初始高度为 0

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x]) # 路径压缩：递归压扁路径
        return self.parent[x] # 返回根节点

    def union(self, x, y):
        rootX = self.find(x) # 查找 x 的根
```

```
        rootY = self.find(y) # 查找 y 的根
        if rootX == rootY:
            return False # 已连通，无需合并

        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY # 小树挂到大树下
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1 # 高度相同时，新树高度 +1
        return True # 合并成功
```

在 find 方法中，递归实现路径压缩，直接将路径节点指向根。union 方法使用秩比较：优先挂接小树，高度相同时更新秩。返回值 True 表示成功合并，便于外部逻辑跟踪。该实现时间复杂度 O(α(n))，空间 O(n)，可直接用于解决算法问题。

8 常见问题答疑（Q&A）

路径压缩和按秩合并可同时使用，因为两者正交：路径压缩优化查询路径，按秩合并优化合并策略；同时应用不会冲突，反而协同降低整体复杂度。秩是否可用节点数量替代？可以，称为重量合并（Union by Size），将小集合挂到大集合下，同样控制树高度；但高度合并（按秩）更精确避免高度暴涨。并查集本身不支持集合分裂；若需分裂操作，需扩展设计如维护反向指针，或改用其他数据结构如 Link-Cut Tree。

本文深入探讨了并查集的核心原理：森林表示法、find/union 操作、双优化策略（路径压缩和按秩合并），以及近常数时间复杂度 O(α(n))。实战中，它高效解决动态连通性问题，如社交网络或图算法。扩展学习建议包括带权并查集（处理关系传递问题，如「食物链」问题中距离权重）、动态并查集（支持删除操作，通过懒标记重建）、或并行并查集算法（分布式系统优化）。掌握这些，读者可进一步挑战复杂场景。