# 最近公共祖先问题 (LCA) 的解决路径

杨锟、李子靖、吴谦、纪芃宇 [1]
(1. 南开大学数学科学学院 300071)

**Abstract**

最近公共祖先（Lowest Common Ancestor, LCA）问题是树形结构中一个核心问题，广泛应用于图论、计算机网络和生物信息学等领域。本文系统综述了解决 LCA 问题的多种经典算法，涵盖了从朴素方法到高级数据结构的完整路径。具体包括：基于 DFS 的朴素算法、二进制提升法、Tarjan 算法、欧拉序结合范围最小值查询（RMQ）方法及其 ±1RMQ 优化，以及动态树场景下的 Link-Cut Tree 算法。每种算法的数学推导、时间复杂度、适用场景和实现细节均被深入分析。通过对比分析，本文为不同应用场景下的 LCA 问题提供了选择依据，并指出了未来优化的潜在方向。

**关键词：**LCA 问题、Tarjan 算法、RMQ 问题、动态树、Link-Cut Tree.

## 1 基本定义

**定义 1.1** 在一颗有根树 $T$ 中，给定两个节点 $u$ 和 $v$，节点 $w \in T$ 是 $u$ 和 $v$ 的**最近公共祖先** *(Lowest Common Ancestor,LCA)*，当且仅当满足以下条件：

1. 祖先条件：$w$ 是 $u$ 和 $v$ 的祖先;
2. $w$ 是 $u$ 和 $v$ 的所有祖先中深度最大的.

LCA（Lowest Common Ancestor，最近公共祖先）问题是在一棵树（通常是二叉树或多叉树）中，寻找两个给定节点的最近公共祖先节点，即在树中同时是这两个节点祖先的、深度最大的节点。我们记点集 $S = \{v_1, v_2, \ldots, v_n\}$ 的最近公共祖先为 $LCA(\{v_1, v_2, \ldots, v_n\})$ 或 $LCA(S)$.

## 2 一些性质

**性质 2.1** $LCA(\{v\}) = v$.

**性质 2.2** $u$ 是 $v$ 的祖先当且仅当 $LCA(u, v) = u$.

**性质 2.3** 如果 $u$ 不为 $v$ 的祖先、$v$ 不为 $u$ 的祖先，那么 $u, v$ 分别处于以 $LCA(\{u, v\})$ 为根节点的两棵不同子树中.

**性质 2.4** 在先序遍历中，$LCA(S)$ 出现在所有点集 $S$ 中的元素之前，后序遍历中 $LCA(S)$ 出现在所有点集 $S$ 中的元素之后.

**性质 2.5** 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $LCA(A \cup B) = LCA(LCA(A), LCA(B))$.

**性质 2.6** 两点的最近公共祖先必定处在连接树上两点间的最短路上.

**性质 2.7** $dis(u, v) = depth(u) + depth(v) - 2depth(LCA(u, v))$，其中 $dis(u, v)$ 是树上两点的距离函数，$depth(u)$ 是点 $u$ 的深度函数.

**Proof** 记 $w = LCA(u, v)$，则 $dis(u, w) = depth(u) - depth(w)$，$dis(v, w) = depth(v) - depth(w)$，故 $dis(u, v) = dis(u, w) + dis(v, w) = depth(u) + depth(v) - 2 \cdot depth(w) = depth(u) + depth(v) - 2depth(LCA(u, v))$. □

# 3 算法实现

## 3.1 朴素算法

### 3.1.1 算法思路

在一棵有根树 $T$ 中，对于给定的节点 $u$ 和 $v$，我们从 $u$ 和 $v$ 分别向上回溯到根 $root$，得到路径 $path_u = \{u, p(u), p(p(u)), ..., root\}$ 和 $path_v = \{v, p(v), p(p(v)), ..., root\}$，其中 $p(u)$ 代表 $u$ 的父节点。由于 $LCA(u, v)$ 是 $path_u \cap path_v$ 的深度最大的几点，我们从路径的末尾开始比较，找到的第一个不同的节点，即为 $u$ 和 $v$ 的最近公共祖先 $LCA(u, v)$.

### 3.1.2 算法实现

**Algorithm 1:** 朴素算法：LCA

```
Function Preprocess(tree, n):
    Initialize parent[n], depth[n] ;                          // 存储父节点和深度
    DFS(0, -1, 0) ;                                  // 从根节点 0 开始，父节点为-1，深度为 0
Function DFS(u, p, d):
    parent[u] ← p;
    depth[u] ← d;
    for each child v of u do
        DFS(v, u, d + 1);
Function LCA(u, v):
    Initialize empty lists path_u, path_v;
    while u ≠ −1 do
        Append u to path_u;
        u ← parent[u];
    while v ≠ −1 do
        Append v to path_v;
        v ← parent[v];
    lca ← −1;
    for i ← min(|path_u|, |path_v|) − 1 downto 0 do
        if path_u[i] == path_v[i] then
            lca ← path_u[i];
            break;
    return lca;
```

### 3.1.3 复杂度分析

我们通过 DFS 计算得到每个节点 $u$ 的父节点 $p(u)$ 和深度 $depth(u)$，这样预处理的复杂度为 $O(n)$. 其中追溯路径的时间复杂度为 $O(n)$，单次查询的复杂度为 $O(n)$. 故对于 $m$ 次查询，总的复杂度为 $O(n + mn)$.

这种朴素算法的特点就是实现方式十分简单，适合小规模树或者查询次数较少的情况. 但是一旦树结构复杂或者需要大规模查询，这个方法就会十分低效.

我们发现，在朴素算法的预处理父节点和深度的过程可以通过一定的优化避免重复的计算，但是在查询上仍然需要遍历整条路径，没法进一步优化.

## 3.2 二进制提升算法

### 3.2.1 算法思路

在一棵有根树 $T$ 中，对于给定的节点 $u$ 和 $v$，我们可以通过预处理得到每个节点的 $2^k$ 级祖先，在查询的时候通过二进制调整 $u$ 和 $v$ 到同一深度，然后逐步向上调整，直到找到 $u$ 和 $v$ 的最近公共祖先.

- **预处理：**我们定义 $anc(u,k)$ 为节点 $u$ 的 $2^k$ 级祖先，即 $anc(u,k)=p^{2^k}(u)$ 其中 $p(u)$ 为 $u$ 的父节点，$p^{2^k}(u)$ 表示对 $u$ 取 $2^k$ 次父节点. 首先显然有 $anc(u,0)=p(u)$，其次由于 $2^k=2^{k-1}+2^{k-1}$，我们可以通过 $2^{k-1}$ 级祖先去计算 $2^k$ 级祖先，即：$anc(u,k)=anc(anc(u,k-1),k-1)$
- **查询：**
  - **深度对齐：**我们将 $u$ 和 $v$ 的深度对齐，即若 $depth(u)>depth(v)$，我们将 $u$ 提升 $d=depth(u)-depth(v)$ 层. 我们将 $d$ 分解为二进制形式，依次需要跳跃 $2^k$ 步.
  - **$LCA$ 查找：**当 $u$ 与 $v$ 深度相同时，若 $u\neq v$ 显然有 $LCA(u,v)\neq u,v$. 我们考虑从大到小跳跃 $2^k$ 步，若 $anc(u,k)\neq anc(v,k)$，则令 $u\leftarrow anc(u,k),v\leftarrow anc(v,k)$. 最终，$u$ 和 $v$ 到达 $LCA(u,v)$ 的直接子节点，$LCA(u,v)$ 为其父节点.

### 3.2.2 算法实现

1. **预处理：**
   - 通过深度优先搜索（DFS）遍历树，计算每个节点的父节点 $p(u)$ 和深度 $\text{depth}(u)$。
   - 构建二维数组 $\text{parent}[u][k]$，存储节点 $u$ 的 $2^k$ 级祖先：
     - 初始化 $\text{parent}[u][0]=p(u)$，即 $u$ 的直接父节点。
     - 对于 $k\geq 1$，使用动态规划计算：

$$\text{parent}[u][k]=\text{parent}[\text{parent}[u][k-1]][k-1]$$

     - 若 $\text{parent}[u][k-1]=-1$（超出根节点），则设置 $\text{parent}[u][k]=-1$。
2. **查询 $LCA$：**
   - **深度对齐：**
     - 若 $\text{depth}(u)<\text{depth}(v)$，交换 $u$ 和 $v$，确保 $u$ 的深度不小于 $v$。
     - 计算深度差 $d=\text{depth}(u)-\text{depth}(v)$。
     - 从 $k=\lfloor\log n\rfloor$ 到 0，检查每个 $k$：若 $2^k\leq d$ 且 $\text{parent}[u][k]\neq -1$，则：更新 $u\leftarrow \text{parent}[u][k]$ 并从 $d$ 中减去 $2^k$。
   - **LCA 查找：**
     - 若 $u=v$，则返回 $u$（即 LCA）。
     - 从 $k=\lfloor\log n\rfloor$ 到 0：若 $\text{parent}[u][k]\neq\text{parent}[v][k]$ 且 $\text{parent}[u][k]\neq -1$，则：更新 $u\leftarrow\text{parent}[u][k]$，$v\leftarrow\text{parent}[v][k]$。
   - 返回 $\text{parent}[u][0]$，即 $u$ 和 $v$ 的父节点（LCA）。

**Algorithm 2:** 二进制提升算法

```
Function Preprocess(tree, n):
    Initialize parent[n][log n], depth[n] ;                    // 存储 2^k 级祖先和深度
    DFS(0, -1, 0) ;                                            // 从根节点 0 开始，父节点-1，深度 0
    for k ← 1 to ⌊log n⌋ do
        for u ← 0 to n − 1 do
            if parent[u][k − 1] ≠ −1 then
                parent[u][k] ← parent[parent[u][k − 1]][k − 1];
            else
                parent[u][k] ← −1;
Function DFS(u, p, d):
    parent[u][0] ← p;
    depth[u] ← d;
    for each child v of u do
        DFS(v, u, d + 1);
Function LCA(u, v):
    if depth[u] < depth[v] then
        Swap u, v ;                                            // 确保 u 深度较大
    for k ← ⌊log n⌋ downto 0 do
        if parent[u][k] ≠ −1 and depth[u] − 2^k ≥ depth[v] then
            u ← parent[u][k] ;                                 // 提升 u 到与 v 相同深度
    if u == v then
        return u;
    for k ← ⌊log n⌋ downto 0 do
        if parent[u][k] ≠ parent[v][k] and parent[u][k] ≠ −1 then
            u ← parent[u][k];
            v ← parent[v][k] ;                                 // 同时提升 u 和 v
    return parent[u][0] ;                                      // LCA 是 u 和 v 的父节点
```

### 3.2.3 复杂度分析

在预处理上，我们通过 DFS 计算得到每个节点的父节点和深度，然后使用动态规划计算每个节点的 $2^k$ 级祖先. 预处理的复杂度为 $O(n \log n)$. 在查询上，深度差最多为 $n$，二进制分解需要 $O(\log n)$ 次跳跃，那么我们从 $\lfloor \log n \rfloor$ 到 0 尝试跳跃，最多需要 $O(\log n)$ 次. 故对于 $k$ 次查询，总的复杂度为 $O(n \log n + k \log n)$.

这种二进制提升算法的查询效率较高 $O(\log n)$，但是预处理的复杂度较高 $O(n \log n)$，适合需要频繁查询的情况，但在大规模树的预处理上耗时较久 [3].

## 3.3 Tarjan 算法

### 3.3.1 算法思路

与前面的算法不同的是，Tarjan 算法是一种离线算法 [1]，即所有查询在算法开始前已知，算法一次性处理所有查询并返回结果.Tarjan 算法利用 DFS 和并查集，通过单次遍历树来回答所有查询.

- **DFS**：从根节点开始进行深度优先搜索，依次访问每个节点及其子树. 当访问完一个节点 $u$ 的所有子树后，$u$ 是其子树中所有节点的祖先.
- **并查集**：并查集维护节点的连通性，初始时每个节点自成一个集合. 当访问完子树 $v$ 后，将 $v$ 的集合合并到其父节点 $u$ 的集合 ($Union(v,u)$). 并查集的代表元 ($Find$ 操作) 就表示子树中节点的“当前祖先”.
- **查询**：对于每个查询 $(u,v)$，当 DFS 访问到 $u$ 且 $v$ 已访问，即 $v$ 的子树已经处理，则 $v$ 所在集合的代表祖先是 $LCA(u,v)$，即 $LCA(u,v) = ancestor[Find(v)]$.

### 3.3.2 算法实现

1. **预处理**：
   - 初始化并查集数组 uf$[u] = u$，表示节点 $u$ 的父集合。
   - 初始化访问标记 visited$[u]$ = false，表示节点 $u$ 是否已访问。
   - 初始化祖先数组 ancestor$[u] = u$，表示节点 $u$ 所在集合的代表祖先。
   - 构建查询邻接表 query$[u]$，存储与节点 $u$ 相关的查询 $(u,v)$。
2. **DFS 遍历与查询处理**：
   - 从根节点 0 开始深度优先搜索（DFS）：
     - 标记节点 $u$ 为已访问：visited$[u]$ = true。
     - 设置 $u$ 为当前子树的代表祖先：ancestor$[u] = u$。
     - 对每个子节点 $v$：将 $v$ 的集合合并到 $u$(Union$(v,u)$)，更新代表祖先：ancestor[Find$(u)$] = $u$。
     - 处理 query$[u]$ 中的查询 $(u,v)$：若 visited$[v]$ = true，则 LCA$(u,v)$ = ancestor[Find$(v)$]。
   - 返回所有查询的 LCA 结果，存储在 result 数组中。

**Algorithm 3:** Tarjan 离线算法：LCA

```
Function Find(x):
    if uf[x] ≠ x then
        uf[x] ← Find(uf[x]) ;                                   // 路径压缩
    return uf[x];
Function Union(x, y):
    uf[Find(x)] ← Find(y) ;                                     // 合并集合
Function TarjanLCA(tree, queries, n):
    Initialize uf[n], ancestor[n], visited[n] ;                 // 并查集、祖先、访问标记
    Initialize result[q] ;                                      // 存储查询结果
    for u ← 0 to n − 1 do
        uf[u] ← u;
        visited[u] ← false;
    Build adjacency list query from queries ;                   // 存储每个节点的查询
    DFS(0, -1) ;                                                // 从根节点 0 开始
    return result;
Function DFS(u, p):
    ancestor[u] ← u ;                                           // 当前节点为祖先
    visited[u] ← true;
    for each child v of u do
        DFS(v, u) ;                                             // 递归访问子节点
        Union(v, u) ;                                           // 合并子树
        ancestor[Find(u)] ← u ;                                 // 更新代表祖先
    for each query (u, v) in query[u] do
        if visited[v] then
            result[query_id] ← ancestor[Find(v)] ;              // 记录 LCA
```

### 3.3.3 复杂度分析

在 DFS 上，访问每个节点和边一次，复杂度为 $O(n)$. 在并查集操作上，$Find$ 和 $Union$ 操作使用路径压缩，均摊的复杂度为 $O(\alpha(n))$，其中 $\alpha(n)$ 是反阿克曼函数，近似于常数，每次访问节点 $u$ 后，可能执行一次 $Union$，复杂度总计为 $O(\alpha(n))$. 对于 $k$ 次查询，每次查询 $(u, v)$ 被处理两次，总计 $O(k)$ 次访问，每次查询均涉及 $O(\alpha(n))$ 的 $Find$ 操作，总查询复杂度为 $O(k\alpha(n))$. 故对于 $n$ 个节点、$k$ 次查询的总复杂度为 $O(n + k\alpha(n))$，近似线性.

该算法适用于任何有根树，无论是二叉树还是多叉树且时间复杂度接近线性，适合大规模输入. 但无法处理在线查询，需要预先知道所有查询，且查询邻接表需要 $O(k)$ 的空间，查询时内存占用大.

### 3.4 欧拉序 +RMQ

#### 3.4.1 相关定义

**定义 3.1 欧拉序：**对于树中的每个节点，我们按照 $DFS$ 的顺序访问节点，记录每次访问节点的顺序，生成一个节点序列.

对于节点 $u$，若其有子节点 $v_1, \ldots, v_m$，则欧拉序包含：第一次访问的 $u$、访问子树 $v_1, \ldots, v_m$、每次从子树返回时再次记录 $u$.

**性质 3.1** 一棵 $n$ 个节点的树，其欧拉序长度为 $2n-1$.

每个节点 $u$ 在欧拉序中至少出现一次，我们使用 $first(u)$ 来代表其第一次出现的位置，用 $e_j$ 表示欧拉序中第 $j$ 位的节点.

**定义 3.2 深度序列：**由欧拉序中每个节点的深度组成的序列.

#### 3.4.2 算法思路

欧拉序 +RMQ（Range Minimum Query，范围最小值查询）适用于静态树（树结构不变）且查询次数较多的场景 [2]。它通过将 LCA 问题转化为 RMQ 问题，利用欧拉序和高效的 RMQ 数据结构实现快速查询。

- **LCA 到 RMQ 的转化：**在欧拉序中，$u$ 和 $v$ 的 $LCA(u,v)$ 是 $[first(u), first(v)]$ 区间内深度最小的节点.
- **RMQ 问题：**给定深度序列 $D = \{depth(e_1), depth(e_2), \ldots, depth(e_{2n-1})\}$，查询区间 $[l,r]$ 的最小值及其对应位置. 换言之，LCA 查询转化为：

$$LCA(u,v) = e_{RMQ(\min(first(u),first(v)),\max(first(u),first(v)))}$$

- **RMQ 问题的解决：**
  - **稀疏表：**我们先进行预处理：构建一个二维表 $st[i][k]$，表示区间 $[i, i+2^k-1]$ 的最小值. 在查询时，我们利用二进制分解，查询 $[l,r]$ 的最小值.
  - **±1RNQ 优化：**在欧拉序中，相邻节点的深度差的绝对值不超过 1. 我们将深度序列 $D$ 进行分块，块的大小为 $\lceil \log n/2 \rceil$，对每个块预计算最小值，利用预计算表和块间最小值，查询任意区间的最小值.

#### 3.4.3 算法实现

1. **预处理欧拉序和深度序列：**
   通过深度优先搜索（DFS）遍历树，生成：
   - 欧拉序 $E = [e_1, e_2, \ldots, e_{2n-1}]$，记录每次访问的节点。
   - 深度序列 $D = [\text{depth}(e_1), \text{depth}(e_2), \ldots, \text{depth}(e_{2n-1})]$，记录对应深度。
   - 每个节点 $u$ 的第一次出现位置 $\text{first}[u]$。
2. **构建 RMQ 数据结构：**
   对深度序列 $D$ 构建范围最小值查询（RMQ）数据结构：
   - 稀疏表方案：预处理时间 $O(n \log n)$，查询时间 $O(1)$。
   - ±1 RMQ 优化：利用深度差不超过 ±1，分块并预计算模式，预处理时间 $O(n)$，查询时间 $O(1)$。

3. **查询 LCA：**

对于查询 LCA$(u, v)$：

- 确定区间 $[l, r] = [\min(\text{first}[u], \text{first}[v]), \max(\text{first}[u], \text{first}[v])]$。
- 使用 RMQ 查询 $D[l..r]$ 的最小值索引 $i$，返回 $e_i$ 作为 LCA。

**Algorithm 4:** 欧拉序 + RMQ (稀疏表方案)

**Function** `Preprocess`(*tree, n*)**:**
  Initialize empty lists `euler`, `depth` ; // 初始化欧拉序和深度序列
  Initialize `first`[$n$] ; // 存储节点的第一次出现位置
  `DFS`(*0, -1, 0*) ; // 从根节点开始 DFS
  `sparse_table` ← `BuildSparseTable`(*depth*) ; // 构建稀疏表
**Function** `DFS`($u$, $p$, $d$)**:**
  `first`[$u$] ← `length`(`euler`) ; // 记录节点第一次出现位置
  Append $u$ to `euler` ; // 添加节点到欧拉序
  Append $d$ to `depth` ; // 添加深度到深度序列
  **for** *each child* $v$ *of* $u$ **do**
    `DFS`($v$, $u$, $d+1$) ; // 递归处理子节点
    Append $u$ to `euler` ; // 返回时再次添加节点
    Append $d$ to `depth` ; // 返回时添加深度
**Function** `BuildSparseTable`(*arr*)**:**
  $n$ ← `length`(`arr`) ; // 获取数组长度
  Initialize `st`[$n$][$\lfloor \log n \rfloor + 1$] ; // 初始化稀疏表
  **for** $i \leftarrow 0$ **to** $n-1$ **do**
    `st`[$i$][0] ← `arr`[$i$] ; // 初始化单点区间
  **for** $k \leftarrow 1$ **to** $\lfloor \log n \rfloor$ **do**
    **for** $i \leftarrow 0$ **to** $n - 2^k$ **do**
      `st`[$i$][$k$] ← $\min($`st`[$i$][$k-1$], `st`[$i + 2^{k-1}$][$k-1$]$)$ ; // 计算更大区间的 RMQ
  **return** `st` ; // 返回稀疏表
**Function** `RMQ`(*st, l, r*)**:**
  $k \leftarrow \lfloor \log_2(r - l + 1) \rfloor$ ; // 计算最大覆盖区间的对数
  **return** $\min($`st`[$l$][$k$], `st`[$r - 2^k + 1$][$k$]$)$ ; // 返回区间最小值
**Function** `LCA`($u$, $v$)**:**
  $l \leftarrow \min($`first`[$u$], `first`[$v$]$)$ ; // 取第一次出现位置的最小值
  $r \leftarrow \max($`first`[$u$], `first`[$v$]$)$ ; // 取第一次出现位置的最大值
  `min_depth` ← `RMQ`(*sparse_table, l, r*) ; // 查询区间最小深度
  **return** `euler[index of min_depth]` ; // 返回对应的 LCA 节点

**Algorithm 5-1:** 欧拉序 + ±1 RMQ（Part-I）

```
Function Preprocess(tree, n):
    Initialize empty lists euler, depth ;                          // 初始化欧拉序和深度序列
    Initialize first[n] ;                                          // 存储节点的第一次出现位置
    DFS(0, -1, 0) ;                                                // 生成欧拉序和深度序列
    rmq_struct ← PreprocessPlusMinusOneRMQ(depth, length(depth)) ; // 构建 ±1 RMQ 结构
    return euler, first, rmq_struct
Function DFS(u, p, d):
    first[u] ← length(euler) ;                                     // 记录节点第一次出现位置
    Append u to euler ;                                            // 添加节点到欧拉序
    Append d to depth ;                                            // 添加深度到深度序列
    foreach child v of u do
        DFS(v, u, d + 1) ;                                         // 递归处理子节点
        Append u to euler ;                                        // 返回时再次添加节点
        Append d to depth ;                                        // 返回时添加深度
Function PreprocessPlusMinusOneRMQ(depth, m):
    block_size ← ⌊log2 m/2⌋ ;                                      // 设置块大小
    num_blocks ← ⌈m/block_size⌉ ;                                  // 计算块数
    Initialize block_min[num_blocks], block_pattern[num_blocks] ;  // 初始化块最小值和模式
    for i ← 0 do
        i ≤ m − 1
    i ← i + block_size  block_idx ← ⌊i/block_size⌋ ;                // 当前块编号
    min_idx ← i ;                                                  // 初始化最小值索引
    min_val ← depth[i] ;                                           // 初始化最小值
    Initialize empty list pattern ;                                // 初始化模式列表
    for j ← i + 1 do
        j ≤ min(i + block_size − 1, m − 1)
    j ← j + 1  if depth[j] < min_val then
        min_idx ← j ;                                              // 更新最小值索引
        min_val ← depth[j] ;                                       // 更新最小值
    Append depth[j] - depth[j − 1] to pattern ;                    // 记录深度差
    block_min[block_idx] ← min_idx ;                               // 存储块最小值索引
    block_pattern[block_idx] ← pattern ;                           // 存储块模式
    pattern_table ← PrecomputePatternTable(block_size) ;           // 预计算模式表
    return block_min, block_pattern, pattern_table ;               // 返回 RMQ 结构
```

**Algorithm 5-2:** 欧拉序 + ±1 RMQ（Part-II）

**Function** `PrecomputePatternTable`(*block_size*)**:**
  Initialize `pattern_table` for all possible patterns ; // 初始化模式表
  **foreach** *possible pattern* $P$ *of length* $\leq$ *`block_size`* *with values* $\pm 1$ **do**
    Initialize `depths[block_size + 1]` with `depths[0]` $\leftarrow 0$ ; // 初始化深度数组
    **for** $i \leftarrow 1$ **do**
      $i \leq |P|$
    $i \leftarrow i+1$ `depths[`$i$`]` $\leftarrow$ `depths[`$i-1$`]` $+ P[i-1]$ ; // 计算相对深度
    **foreach** *subinterval* $[l, r]$ *in* $[0, |P|]$ **do**
      Compute minimum depth and its index in `depths[`$l..r$`]` ; // 计算区间最小值
      Store in `pattern_table[`$P$`][`$l$`][`$r$`]` ; // 存储结果
  **return** `pattern_table` ; // 返回模式表
**Function** `RMQPlusMinusOne`(*rmq_struct, l, r*)**:**
  Extract `block_min`, `block_pattern`, `pattern_table` from `rmq_struct` `block_size` $\leftarrow \lfloor \log_2 m/2 \rfloor$ ; // 块大小
  `start_block` $\leftarrow \lfloor l/$`block_size`$\rfloor$ ; // 起始块
  `end_block` $\leftarrow \lfloor r/$`block_size`$\rfloor$ ; // 结束块
  Initialize `min_idx` $\leftarrow l$, `min_val` $\leftarrow$ `depth[`$l$`]` ; // 初始化最小值和索引
  **if** *`start_block`* $=$ *`end_block`* **then**
    `pattern` $\leftarrow$ `block_pattern[start_block]` ; // 获取块模式
    `rel_l` $\leftarrow l -$ `start_block` $\times$ `block_size` ; // 相对左端点
    `rel_r` $\leftarrow r -$ `start_block` $\times$ `block_size` ; // 相对右端点
    `idx` $\leftarrow$ `pattern_table[pattern][rel_l][rel_r]` ; // 查询模式表
    **return** `start_block` $\times$ `block_size` $+$ `idx` ; // 返回绝对索引
  **for** $i \leftarrow l$ **do**
    $i \leq \min((\texttt{start\_block}+1) \cdot \texttt{block\_size} - 1, r)$
  $i \leftarrow i+1$ **if** *`depth[i]`* $<$ *`min_val`* **then**
    `min_idx` $\leftarrow i$ ; // 更新最小值索引
    `min_val` $\leftarrow$ `depth[`$i$`]` ; // 更新最小值
  **for** $i \leftarrow$ *`end_block`* $\cdot$ *`block_size`* **do**
    $i \leq r$
  $i \leftarrow i+1$ **if** *`depth[i]`* $<$ *`min_val`* **then**
    `min_idx` $\leftarrow i$ ; // 更新最小值索引
    `min_val` $\leftarrow$ `depth[`$i$`]` ; // 更新最小值
  **for** *`block`* $\leftarrow$ *`start_block+1`* **do**
    `block` $\leq$ `end_block` $- 1$
  `block` $\leftarrow$ `block` $+ 1$ **if** *`depth[block_min[block]]`* $<$ *`min_val`* **then**
    `min_idx` $\leftarrow$ `block_min[block]` ; // 更新最小值索引
    `min_val` $\leftarrow$ `depth[block_min[block]]` ; // 更新最小值
  **return** `min_idx` ; // 返回最小值索引

**Algorithm 5-3:** 欧拉序 + ±1 RMQ（Part-III）

```
Function LCA(euler, first, rmq_struct, u, v):
    l ← min(first[u], first[v]) ;                    // 取第一次出现位置的最小值
    r ← max(first[u], first[v]) ;                    // 取第一次出现位置的最大值
    min_idx ← RMQPlusMinusOne(rmq_struct, l, r) ;    // 查询区间最小深度索引
    return euler[min_idx] ;                          // 返回 LCA 节点
```

#### 3.4.4 算法分析

稀疏表和 ±1RMQ 方案的区别主要在于预处理的复杂度，其中稀疏表方案的预处理时间复杂度为 $O(n \log n)$，查询时间复杂度为 $O(1)$;±1RMQ 方案的预处理时间复杂度为 $O(n)$，查询时间复杂度为 $O(1)$，但需要额外的空间存储模式表和块间最小值. 即对于 $n$ 个节点，$k$ 次查询，稀疏表方案的复杂度为 $O(n \log n + k)$，±1RMQ 方案的复杂度为 $O(n + k)$[4].

## 4 LCA 问题的具体应用

最近公共祖先（LCA）问题作为一个基础的树形结构问题，在多个领域具有广泛的应用。以下从几个主要方面探讨其具体应用场景：

### 4.1 图论与算法设计

在图论中，LCA 常用于解决树上路径相关的问题。例如，在计算两节点间的最短路径时，LCA 可用于确定路径的公共部分，从而优化路径查询的效率 [7]。例如，给定树上两节点 $u$ 和 $v$，其最短路径长度可以通过公式 $\text{distance}(u, v) = \text{depth}(u) + \text{depth}(v) - 2 \cdot \text{depth}(\text{LCA}(u, v))$ 计算。此外，LCA 在网络流、树上差分等算法中也起到关键作用，例如用于处理动态更新的树结构问题.

### 4.2 计算机网络

在计算机网络中，LCA 被广泛应用于路由优化和网络拓扑分析。例如，在分层网络（如互联网的自治系统）中，LCA 可用于确定两个节点之间的最近共同路由节点，从而优化数据包的转发路径。此外，在分布式系统中，LCA 可用于分析节点间的通信层次结构，减少消息传递的延迟.

### 4.3 生物信息学

在生物信息学中，LCA 常用于系统发育树的分析 [6]。例如，在研究物种的进化关系时，LCA 可用于确定两个物种的最近共同祖先，从而推断它们的进化分化时间。此外，在基因组学中，LCA 可用于分析基因序列的层次结构，帮助构建基因家族的谱系树.

### 4.4 数据库与信息检索

在数据库系统中，LCA 可用于优化基于层次结构的查询 [8]。例如，在 XML 或 JSON 数据结构的查询中，LCA 可用于快速定位两个元素的共同父节点，从而提高查询效率。此外，在信息检索中，LCA 可用于分析文档的语义结构，辅助语义搜索.

## 5 总结

本文系统地分析了解决最近公共祖先（LCA）问题的多种算法，从朴素的基于 DFS 的方法到高效的欧拉序结合 ±1 RMQ 方案，以及动态场景下的 Link-Cut Tree 算法。每种算法在时间复杂度、空间复杂度和适用场景上各有优劣。例如，朴素算法实现简单但查询效率较低，适合小规模问题；二进制提升法和欧拉序 +RMQ 方法在静态树上具有高效的查询性能，适合大规模在线查询；Tarjan 算法适合离线场景；而 Link-Cut Tree 则适用于动态树结构。

通过对比分析，我们可以为不同应用场景选择合适的算法。例如，在静态树的大规模查询中，欧拉序 +RMQ 方法以其 $O(n)$ 预处理和 $O(1)$ 查询的性能表现最佳；在动态场景下，Link-Cut Tree 提供了灵活的解决方案。未来研究方向可以包括：进一步优化动态 LCA 算法的空间复杂度，探索基于并行计算的 LCA 算法，以及在分布式系统中实现高效的 LCA 查询。

## References


[1] R. E. Tarjan, "Applications of path compression on balanced trees," *Journal of the ACM*, vol. 26, no. 4, pp. 690–715, 1979.

[2] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," *LATIN 2000: Theoretical Informatics*, pp. 88–94, 2000.

[3] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[4] J. Fischer and V. Heun, "Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE," *Combinatorial Pattern Matching*, pp. 36–48, 2006.

[5] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences*, vol. 26, no. 3, pp. 362–391, 1983.

[6] D. Gusfield, "Algorithms on strings, trees, and sequences: Computer science and computational biology," *Cambridge University Press*, 1997.

[7] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification—a technique for speeding up dynamic graph algorithms," *Journal of the ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[8] N. Alon and B. Schieber, "Optimal preprocessing for answering on-line product queries," *Technical Report*, Tel Aviv University, 1994.