

# 二次时间内求解循环最长公共子序列

Andy Nguyen\*

October 18, 2023

## Abstract

我们提出了一个实用的算法来解决循环最长公共子序列(C LCS)问题, 其运行时间为 $O(mn)$ , 其中 $m$ 和 $n$ 是两个输入字符串的长度。虽然这并不一定是对现有记录的渐进改进, 但它更容易理解和实现。

## 1 简介

最长公共子序列(LCS)问题是一个经典问题, 其广泛应用于从DNA序列比对的字符串匹配到形状距离的动态时间规整等领域。虽然原始问题将具有开始和结束的字符串视为输入, 但自然的扩展是考虑循环字符串, 其中起始点在两个字符串上都不固定。这种扩展对于匹配DNA质粒和闭合曲线等对象是有用的。本文给出了一个求解循环最长公共子序列(C LCS)问题的实用算法, 该算法的运行时间为 $O(mn)$ , 其中 $m$ 和 $n$ 是两个输入字符串的长度。

## 2 相关工作

最先进的普通算法LCS来自[4], 他们应用标准的动态规划算法与“四俄罗斯”加速[1], 以实现 $O(n^2/\lg n)$ 的运行时间。也有一些算法(cite)根据不同的输入获得了更好的运行时间, 但所有这些算法在最坏情况下都渐进地变慢。

LCS的循环版本最近有了更多的改进。第一个重大改进来自[3], 它在 $O(mn \lg m)$ 运行时使用分而治之的策略来解决问题。从那时起, 这种方法被应用于问题的泛化, 并发现了实际的优化;例如[5]。我们在初步研究中描述了这些论文中使用的设置, 但我们在此基础上构建了一种不同的方法来实现我们的新算法。[2]提供了在 $O(n^2)$ 时间内运行任意输入的解决方案, 在类似输入上具有渐近更好的性能;然而, 这个解决方案需要大量的机制来理解和实现。

## 3 开场白

给定长度为 $n$ 的字符串 $A$ 和整数 $k$ ,  $0 \leq k < n$ , 我们可以定义 $cut(A, k) = substring(A, k, n) + substring(A, 0, k)$ 。例如,  $cut("abcd", 2) = "cd" + "ab" = "cdab"$ , 而 $cut("abcd", 0) = "abcd" + "" = "abcd"$ 。通过在 $k < 0$ 或 $k \geq n$ 时定义 $cut(A, k) = cut(A, k \bmod n)$ , 我们可以将此定义扩展到任何整数 $k$ 。

让我们定义以下内容:

**Definition 1.** Let  $A$  and  $B$  be strings of length  $m$  and  $n$  respectively, where  $m \leq n$ . A cyclic longest common subsequence of  $A$  and  $B$   $CLCS(A, B)$  is a string that satisfies the following properties:

1.  $CLCS(A, B) = LCS(cut(A, i), cut(B, j))$  for some integers  $i$  and  $j$ .
2.  $|CLCS(A, B)| \geq |LCS(cut(A, i), cut(B, j))|$  for all integers  $i$  and  $j$ .

---

\*斯坦福大学计算机科学系

我们首先可以观察到 $CLCS$ 并不比 $LCS$ 容易。为了看到这一点，我们观察到我们可以使用 $CLCS$ 通过在 $A$ 和 $B$ 前面加上 $m$ 个 $x$ 来解决 $LCS(A, B)$ ，其中 $x$ 是两个字符串中都没有找到的符号。这迫使 $CLCS$ 保持 $A$ 和 $B$ 彼此对齐，以匹配所有 $x$ ，允许我们从结果中提取 $LCS(A, B)$ 。

另一方面，我们也可以表明 $CLCS$ 并不比 $LCS$ 难多少。我们可以通过解决 $LCS$ 的 $mn$ 实例来解决 $CLCS$ ，对应于字符串的每一对可能的切割 $A$ 和 $B$ 。这意味着我们有一个可以在 $O(m^2n^2)$ 时间内运行的 $CLCS$ 解决方案。事实上，我们还可以做得更好：

**Lemma 1.** 存在一个循环最长公共子序列 $A$ 和 $B$ 的 $CLCS(A, B)$ ，它满足以下性质：

1.  $CLCS(A, B) = LCS(cut(A, k), B)$  为某个整数 $k$ 。
2.  $|CLCS(A, B)| \geq |LCS(cut(A, i), cut(B, j))|$  对于所有整数 $i$ 和 $j$ 。

*Proof.* 请参阅[3]。 □

这个引理允许我们只运行 $LCS$ 的 $m$ 实例，给我们一个 $O(m^2n)$ 的运行时间。现在我们看看能不能做得更好。为此，让我们稍微绕一圈，看看 $regular\ LCS$ 。回想一下，我们可以使用存储长度 $len(i, j)$ 和父指针 $parent(i, j)$ 的二维动态规划(DP)表在 $O(mn)$ 时间内解决 $LCS(A, B)$ 。通常我们把这个表看成一个二维数组；然而，我们也可以将此表视为一个有向网格图 $G_{A,B}$ ，如下所示(图1)：

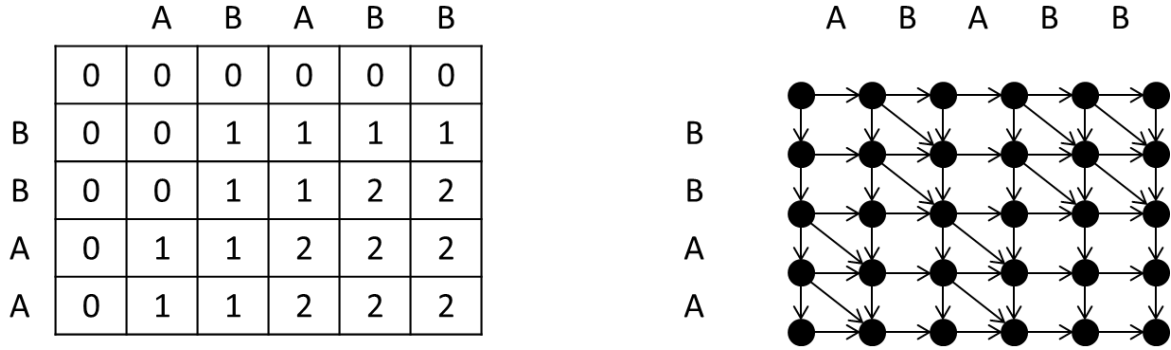


Figure 1: 将DP表可视化网格图。

- 对于表中的每个条目 $e_v$ ，我们有一个节点 $v$ 。
- 在计算 $e_v$ 时考虑到 $e_u$ 的表中，对于每个相邻的条目对 $e_u$ 和 $e_v$ ，我们有一条从 $u$ 到 $v$ 的有向边。请注意，所有的边都指向下方或右侧，因此该图是非循环的。我们可以看到，图中出现了所有水平和垂直的边缘，此外，两个字符串中匹配的对应字符都有对角线边缘。

为了方便起见，我们将固定图形的方向，如图1所示，即 $(0,0)$ 为左上角， $(m,n)$ 为右下角。在这个方向上，第一个坐标表示行，第二个坐标表示列。

**Lemma 2.** 查找 $LCS(A, B)$ 等价于在 $G_{A,B}$ 中查找从 $(0,0)$ 到 $(m,n)$ 的最短路径。

*Proof.* 请参阅[3]。 □

**Corollary 1.** The set of parent pointers computed by any DP solution to  $LCS(A, B)$  (when treated as undirected edges) forms a shortest path tree of  $G_{A,B}$  rooted at  $(0,0)$ .

请注意，在并列情况下，无论父指针的选择如何，这都是正确的。

这意味着我们可以通过解决具有单位边长度的有向无环图上的最短路径问题来解决 $LCS$ 。

## 4 算法

根据引理2, 我们可以通过解决 $m$ 最短路径问题来解决 $CLCS$ ,  $A$ 的每一截对应一个。这些最短路径问题密切相关, 正如我们在构建图 $G_{AA,B}$ (图2)时所看到的。然后, 我们要找到的最短路径是从 $(0,0)$ 到 $(m,n)$ , 从 $(1,0)$ 到 $(m+1,n)$ , 从 $\dots$ , 从 $(m-1,0)$ 到 $(2m-1,n)$ 。现在, 我们还没有做任何事情来节省计算时间; 找到这些 $m$ 最短路径仍然会花费我们 $O(m^2n)$ 时间。然而, 由于这些路径都位于同一个图上, 我们希望可以在最短路径计算之间重用一些工作。

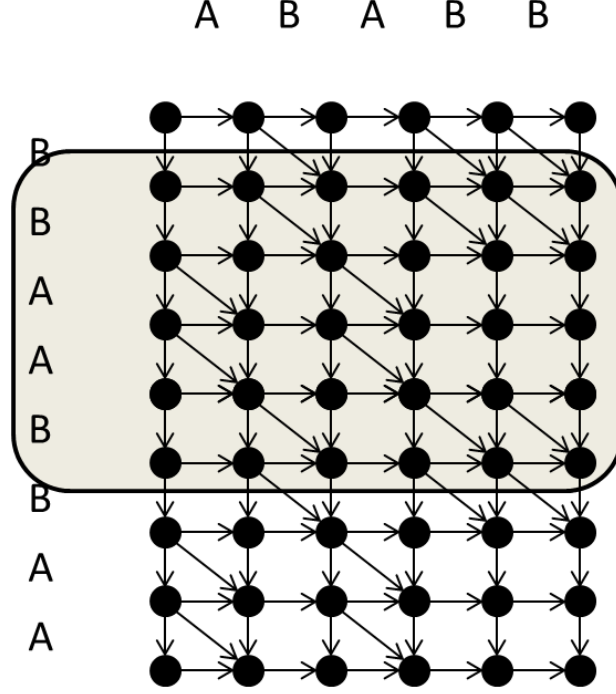


Figure 2: 构造图形 $G_{AA,B}$ 。

由于我们已经固定了图的方向, 我们可以就这个方向称一条边比另一条边低, 对于路径也可以这样做。然后定义以下内容:

**Definition 2.** A lowest shortest path tree of  $G_{A,B}$  is a shortest path tree where for every node  $v$ , the path from the root to  $v$  contained in the shortest path tree is lower than any other shortest path from the root to the node.

首先, 我们展示如何高效地计算最小最短路径树。

**Lemma 3.**  $DP$ 解决方案计算出的父指针集合首先倾向于 $\leftarrow$ , 然后是 $\nwarrow$ , 最后是 $\uparrow$ (在并列情况下), 形成一个以 $G_{A,B}$ 为根的最小最短路径树 $(0,0)$ 。

*Proof.* 假设有一个矛盾: 在某个节点 $v$ 中, 从 $(0,0)$ 到 $v$ 的最短路径 $p'$ 比包含在最短路径树中的最短路径 $p$ 更低。从 $v$ 开始, 回溯到 $(0,0)$ ,  $p'$ 和 $p$ 必须最终在某个节点 $w$ 分叉。这种差异意味着 $p'$ 和 $p$ 跟在两个不同的有效父指针之后。但是, 由于 $p$ 跟随首先倾向于 $\leftarrow$ 的父指针, 然后是 $\nwarrow$ , 最后是 $\uparrow$ , 因此可以得出 $p'$ 不能低于 $p$ 。□

这个引理意味着, 在局部维护 $DP$ 打破平局的策略会产生一个全局最小的最短路径树。我们将利用这一点来限制我们需要做的工作, 以提取所有 $m$ 最短路径。

如果我们计算 $G_{AA,B}$ 的最低最短路径树, 我们可以立即读出从 $(0,0)$ 到 $(m,n)$ 的最短路径。在我们可以读取从 $(1,0)$ 到 $(m+1,n)$ 的最短路径之前, 我们需要在 $(1,0)$ 重新根化最低的最短路径树。我们可以看到, 这相当于完全删除第一行, 然后修复树的其余部分以恢复其属性。

**Lemma 4.** 如果我们从根在 $(0,0)$ 的最低最短路径树中删除最上面一行节点，我们最多只剩下两棵子树 $L$ 和(可能) $R$ 。

*Proof.* 我们通过下面的例子来说明这一点:树的第一行和其余部分之间最多只有两条边。最左边的垂直边缘，从 $(0,0)$ 到 $(1,0)$ ，必须清楚地地在树中，我们表示以 $(1,0)$ 为根的子树为 $L$ 。不可能有其他的垂直边，因为最左边的垂直边为我们提供了一条到第二行的任何节点的较低路径。现在考虑第一个对角线，从左到右扫描，从 $(0,k)$ 到 $(1,k+1)$ 。我们将根在 $(1,k+1)$ 的子树表示为 $R$ 。同理，在这条边的右边也不可能有其他对角线。  $\square$

注意，由于我们对最低最短路径树使用的平局规则， $R$ 中的任何节点与 $L$ 中的节点相邻(向左或对角向上和向左;我们在 $R$ 中称这些节点为边界节点)选择其父节点在 $R$ 中，因为该选择严格优于 $L$ 中的父选项。

接下来，我们注意到DP表中 $LCS(AA,B)$ 对应的长度项(回想一下，这些是公共子序列的长度，而不是图中路径的长度):

**Lemma 5.** 对于表中的任何条目 $(i,j)$ ,  $0 \leq \text{len}(i,j) - \text{len}(i,j-1) \leq 1$ 。

*Proof.* 显然 $\text{len}(i,j) \geq \text{len}(i,j-1)$ 。现在假设有矛盾 $\text{len}(i,j) - \text{len}(i,j-1) > 1$ 。我们从 $(i,j)$ 开始，沿着父指针回溯，直到输入列 $j-1$ 。这样做时，我们将找到一个条目，其长度要么是 $\text{len}(i,j)$ (如果跟在 $\leftarrow$ 后面)，要么是 $\text{len}(i,j)-1$ (如果跟在 $\searrow$ 后面)。无论如何，这个条目在同列中高于 $(i,j-1)$ ，但严格来说要大一些，因此我们可以将这个条目直接填充到 $(i,j-1)$ ，这违背了 $\text{len}(i,j-1)$ 的最优性。  $\square$

这个引理结合我们之前对 $L$ 和 $R$ 的观察，可以得出以下关键事实:

**Corollary 2.** Let  $(i,j)$  be such that  $(i,j) \in R$  but  $(i,j-1) \in L$ . Then  $\text{len}(i,j) = \text{len}(i,j-1) + 1$ .

现在，如果我们通过设置 $\text{parent}(1,k+1) = \leftarrow$ 重新连接两个子树，我们将把 $R$ 中所有节点的长度值降低到1(因为我们失去了 $\text{parent}(1,k+1)$ 的对角线)。但是，由于每个边界节点的旧父节点的选择严格优于 $L$ 中的选项，降低 $R$ 中所有节点的长度值不会导致长度表中的任何不一致，这意味着我们有一个最短路径树。这并不一定是最小最短路径树;然而，推论2告诉我们，在此重新连接之后，每个边界节点 $(i,j) \in R$ 满足 $\text{len}(i,j) = \text{len}(i,j-1)$ ，这意味着在最低最短路径树中 $\text{parent}(i,j) = \leftarrow$ 。此外， $L$ 中的每个节点的长度都保持不变，并且其潜在的父节点都没有获得长度，因此其父指针保持不变。类似地， $R$ 中的每个非边界节点都会丢失一个长度单位，其所有潜在的父节点也是如此，因此其父指针也不会改变。

这意味着如果我们将每个边界节点的父节点设置为 $\leftarrow$ ，我们现在有一个最低的最短路径树，根在 $(1,0)$ 。请注意，我们可以沿着图遍历，通过向下或向右移动(或同时向右移动)来找到所有边界节点，这意味着我们可以在 $O(m+n) = O(n)$ 时间内执行此重定向过程。伪代码如下所示，我们假设输入树的根在 $(\text{root}-1,0)$ 。

```

RE-ROOT(root, m, n, parent)
1   i = root
2   j = 1
3   while j ≤ n and parent[i, j] ≠ ↖
4       j = j + 1
5   if j > n
6       return
7   parent[i, j] = ←
8   while i < 2m and j < n
9       if parent[i + 1, j] == ↑
10          i = i + 1
11          parent[i, j] = ←
12      elseif parent[i + 1, j + 1] == ↖
13          i = i + 1
14          j = j + 1
15          parent[i, j] = ←
16      else
17          j = j + 1
18  while i < 2m and parent[i + 1, j] == ↑
19      i = i + 1
20      parent[i, j] = ←

```

有了这个过程，我们现在可以通过计算 $G_{AA,B}$ 的最低最短路径树从 $G_{AA,B}$ 提取 $m$ 最短路径，然后交替从根读取到适当的结束节点的最短路径，并重新根化较低一个节点的最低最短路径树。我们只需要执行 $m$ 次，这意味着整个过程在 $O(mn)$ 时间内运行。伪代码如下所示。

```

CLCS(A, B)
1   m = A.length
2   n = B.length
3   let len[0...2m, 0...n] and parent[0...2m, 0...n] be new tables
4   LCS(AA, B, len, parent)
5   S = TRACE-LCS(m, n, A, parent)
6   for i = 1 to m - 1
7       RE-ROOT(i, m, n, parent)
8        $\hat{S}$  = TRACE-LCS(m + i, n, A, parent)
9       if S.length <  $\hat{S}$ .length
10          S =  $\hat{S}$ 
11  return S

```

## 5 实现

如果我们需要返回子序列字符串，我们必须遵循上面的伪代码。如果我们只需要返回字符串的长度，我们可以跳过`traceback`子例程，而是简单地从长度表中读取与`traceback`开始对应的条目。在这种情况下，我们需要在每次重根后更新长度表的远边，将长度减1。除了简化代码之外，这一修改还降低了算法的缓存效率低下问题，从而在实践中提高了运行时间。

值得注意的是，实现该算法所需的工作量远远小于[3]中给出的 $O(mn \lg m)$ 算法或[2]中给出的 $O(n^2)$ 算法。

## 6 结论

本文给出了一个在 $O(mn)$ 时间内解决 $CLCS$ 问题的实用算法。请注意，正确性证明在很大程度上依赖于图中所有边的权重都为1的事实。这意味着该算法不能自然地应用于问题的常见扩展，如编辑距离和动态时间弯曲。该算法是否适用于某些类别的边权重，可能值得研究。

## References

- [1] ARLAZAROV, V. L., DINIC, E. A., KRONROD, M. A., AND FARADŽEV, I. A. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics—Doklady* 11, 5 (1970), 1209–1210.
- [2] LANDAU, G. M., MYERS, E. W., AND SCHMIDT, J. P. Incremental string comparison. *SIAM J. Comput.* 27, 2 (Apr. 1998), 557–582.
- [3] MAES, M. On a cyclic string-to-string correction problem. *Inf. Process. Lett.* 35, 2 (June 1990), 73–78.
- [4] MASEK, W. J., AND PATERSON, M. S. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences* 20, 1 (1980), 18 – 31.
- [5] SCHMIDT, F., FARIN, D., AND CREMERS, D. Fast matching of planar shapes in sub-cubic runtime. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (oct. 2007), pp. 1 –6.