Computing a Longest Common Palindromic Subsequence

Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman

AℓEDA group
Department of CSE, BUET, Dhaka - 1000, Bangladesh
{shihab,mahbub86,sumaiya,msrahman}@cse.buet.ac.bd

Abstract. The *longest common subsequence (LCS)* problem is a classic and well-studied problem in computer science. Palindrome is a string, which reads the same forward as it does backward. The *longest common palindromic subsequence (LCPS)* problem is an interesting variant of the classic LCS problem which finds the longest common subsequence between two given strings such that the computed subsequence is also a palindrome. In this paper, we study the LCPS problem and give efficient algorithms to solve this problem. To the best of our knowledge, this is the first attempt to study and solve this interesting problem.

Keywords: longest common subsequence, palindromes, dynamic programming, range query

1 Introduction

The *longest common subsequence (LCS)* problem is a classic and well-studied problem in computer science with a lot of variants arising out of different practical scenarios. In this paper, we introduce and study the *longest common palindromic subsequence (LCPS)* problem: given a pair of strings X and Y over the alphabet Σ, the goal of the LCPS problem is to compute a LCS Z of X and Y such that, Z is a palindrome. In what follows, for the sake of convenience we will assume, that X and Y have equal length, n. But our result can be easily extended to handle two strings of different length.

String and sequence algorithms related to palindromes have attracted stringology researchers since long [2, 4, 6–8]. The LCPS problem also seems to be a new interesting addition to the already rich list of problems related to palindromes. To the best of our knowledge, there exists no research work in the literature on computing longest common palindromic subsequences. However, the problem of computing palindromes and variants in a single sequence has received much attention in the literature. Manacher discovered an on-line sequential algorithm that finds all *'initial'*[1] palindromes in a string [7]. Gusfield gave a linear-

[1] A string $X[1 \dots n]$ is said to have an initial palindrome of length k if the prefix $S[1 \dots k]$ is a palindrome.

time algorithm to find all *'maximal' palindromes* in a string [3]. Authors in [8] solved the problem of finding all palindromes in SLP (Straight Line Programs)-compressed strings. Very recently, Tomohiro *et. al.* worked on pattern matching problems involving palindromes [9].

In this paper, we propose two methods for finding an LCPS, given two strings. Firstly we present a dynamic programming algorithm to solve the problem with time complexity $O(n^4)$, where n is the length of the strings (Section 3). Then, we present another algorithm that runs in $\mathcal{O}(\mathcal{R}^2 \log^2 n \log\log n)$ time (Section 4). Here, the set of all ordered pair of matches between two strings is denoted by $\mathcal{M}$ and $|\mathcal{M}| = \mathcal{R}$. Due to space constraints all the proofs are omitted.

2 Preliminaries

We assume a finite alphabet, Σ. For a string X, we denote its substring $x_i \dots x_j$ $(1 \leq i \leq j \leq n)$ by $X_{i,j}$. For two strings X and Y, if a common subsequence Z of X and Y is a palindrome, then Z is said to be a *common palindromic subsequence (CPS)*. A *CPS* of two strings having the maximum length is called the *Longest Common Palindromic Subsequence (LCPS)* and we denote it by $LCPS(X, Y)$. The set of all matches between two strings X and Y is denoted by $\mathcal{M}$ and it is defined as, $\mathcal{M} = \{(i, j) : 1 \leq i \leq n, 1 \leq j \leq n \text{ and } x_i = y_j\}$. And we have, $|\mathcal{M}| = \mathcal{R}$. We define, $\mathcal{M}_\sigma$ as a subset of $\mathcal{M}$ such that all matches within this set match to a single character $\sigma \in \Sigma$. We have $|\mathcal{M}_\sigma| = \mathcal{R}_\sigma$. Each member of $\mathcal{M}_\sigma$ is called a *σ-match*.

3 A Dynamic Programming Algorithm

We observe that the natural classes of sub-problems for LCPS correspond to pairs of *substrings* of the two input sequences. Based on this observation we present the following theorem which proves the optimal substructure property of the LCPS problem.

Theorem 1. *Let X and Y are two sequences of length n, and $X_{i,j}$ and $Y_{k,\ell}$ are two substrings of X and Y respectively. Let $Z = z_1 z_2 \dots z_u$ be the* LCPS *of the two substrings, $X_{i,j}$ and $Y_{k,\ell}$. Then, the following statements hold,*

1. *If $x_i = x_j = y_k = y_\ell = a$ $(a \in \Sigma)$, then $z_1 = z_u = a$ and $z_2 \dots z_{u-1}$ is an* LCPS *of $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$.*
2. *If $x_i = x_j = y_k = y_l$ condition does not hold then, Z is an* LCPS *of ($X_{i+1,j}$ and $Y_{k,\ell}$) or ($X_{i,j-1}$ and $Y_{k,\ell}$) or ($X_{i,j}$ and $Y_{k,\ell-1}$) or ($X_{i,j}$ and $Y_{k+1,\ell}$).*

Based on Theorem 1 we give the following recursive formulation for the length of $LCPS(X, Y)$:

$$lcps[i,j,k,\ell] = \begin{cases} 0 & i > j \text{ or } k > \ell \\ 1 & (i = j \text{ and } k = \ell) \\ & \text{and} \\ & (x_i = x_j = y_k = y_\ell \\ 2 + lcps[i+1, j-1, k+1, \ell-1] & (i < j \text{ and } k < \ell) \\ & \text{and} \\ & x_i = x_j = y_k = y_\ell \\ \max(lcps[i+1,j,k,\ell], lcps[i,j-1,k,\ell], & \\ lcps[i,j,k+1,\ell], lcps[i,j,k,\ell-1]) & (i \leq j \text{ and } k \leq \ell) \\ & \text{and} \\ & (x_i = x_j = y_k = y_\ell) \\ & \text{does not hold} \end{cases} \quad (1)$$

$lcps[i, j, k, \ell]$ is the length of the *LCPS* of $X_{i,j}$ and $Y_{k,\ell}$. The length of $LCPS(X, Y)$ will be stored at $lcps[1, n, 1, n]$. We can compute this length in $\mathcal{O}(n^4)$ time using a bottom up dynamic programming.

4 A Second Approach

We shall first reduce our problem to a geometry problem and then solve it with the help of modified version of range tree data structure. First, we make the following claim.

Claim 1 *Any common palindromic subsequence $Z = z_1 z_2 \dots z_u$ of two strings X and Y can be decomposed into a set of σ-match pairs ($\sigma \in \Sigma$).*

It follows from Claim 1 that constructing a CPS of X and Y can be seen as constructing an appropriate set of σ-match pairs between them. An arbitrary σ-match pair, $\langle (i, k), (j, \ell) \rangle$ (say m_1), from among all σ-matche pairs between X and Y, can be seen as inducing a substring pair in them. Now we want to construct a CPS Z with length u, placing m_1 at the two ends of Z. Clearly we have $z_1 = z_u = x_i = x_j = y_k = y_\ell$. To compute Z, we will have to recursively select σ-match pairs between $X_{i,j}$ and $Y_{k,\ell}$. This will yield a set of σ-matche pairs corresponding to Z. If we consider all possible σ-match pairs as the two end points of Z, then the longest one obtained will be an *LCPS* of X and Y. Each match between X and Y can be visualized as a point on a $n \times n$

rectangular grid with integer coordinates. Any σ-match pair defines two corner points of a rectangle and thus induces a rectangle in the grid. Now, our goal is to take a σ-matche pair as the two ends of a CPS and recursively construct the set of σ-matche pairs from within the induced rectangle. In particular we take the following steps to compute $LCPS(X, Y)$:

1. Identify an induced rectangle (say Ψ_1) by a pair of σ-matches. Then, pair up σ-matches within Ψ_1 to obtain another rectangle (say Ψ_2) and so on until we encounter either of the following two terminating conditions:
   - T1. If there is no point within any rectangle. This corresponds to the case when there is no match between the substrings.
   - T2. If it is not possible to take any pair of σ-matches within any rectangle. In this case we pair a match with itslef, it corresponds to the single character case in our Dynamic Programming solution.
2. We repeat the above step for all possible σ-match pairs ($\forall\sigma \in \Sigma$). At this point, we have a set of nested rectangle structures. An increase in the nesting depth of the rectangle structures as it is being constructed, corresponds to adding a pair of symbols[2] to the resultant palindromic subsequence. Hence, the set of rectangles with maximum nesting depth gives us an *LCPS*.

Now the problem reduces to the following interesting geometric problem: *Given a set of nested rectangles defined by the σ-match pairs $\forall\sigma \in \Sigma$, we need to find the set of rectangles having the maximum nesting depth.* We refer to this problem as the Maximum Depth Nesting Rectangle Structures (MDNRS) problem.

We assume, without the loss of generality that (i, k) and (j, ℓ) correspond to the lower left corner and upper right corner of the rectangle $\Psi\langle(i, k), (j, \ell)\rangle$. Now, a rectangle $\Psi'(\langle(i', k'), (j', \ell')\rangle)$ will be nested within rectangle $\Psi(\langle(i, k), (j, l)\rangle)$ *iff* the following condition holds:

$i' > i$ and $k' > k$ and $j' < j$ and $\ell' < \ell \Leftrightarrow (i', k', -j', -\ell') > (i, k, j, \ell)$.

Now we convert a rectangle $\Psi(\langle(i, k), (j, \ell)\rangle)$ to a 4-D point $P_\Psi(i, k, -j, -\ell)$ and say that, a point (x, y, z, w) is chained to another point (x', y', z', w') *iff* $(x, y, z, w) > (x', y', z', w')$. Then, a rectangle $\Psi'(\langle(i', k'), (j', \ell')\rangle)$, is nested within a rectangle $\Psi(\langle(i, k), (j, \ell)\rangle)$ *iff* the point $P_{\Psi'}(i', k', -j', -\ell')$ is chained to the point $P_\Psi(i, k, -j, -\ell)$. Hence, the MDNRS problem in 2-D reduces to finding the set of corresponding points in 4-D having the maximum chain length.We refer to this problem as Maximum Chain Length (MCL) Problem. We solve the MCL problem in $O(\mathcal{R}^2 \log^2 n \log\log n)$ time using a modified version of 3-D range tree data structure [1]. A d-dimension range tree, $\mathcal{T}$ is in the form of multi-level trees using an inductive definition on d. Any update and query operation in $\mathcal{T}$ can be done in $\mathcal{O}(\log^d n)$ time. So in 3-D, our query and update performs

[2] If condition T2 is reached, only a symbol shall be added.

in $O(\log^3 n)$ time where the array is of $n \times n \times n$ size. We process the 4-D points (x, y, z, w) in non-increasing order of the highest dimension w. For each point (x, y, z, w) we query in $\mathcal{T}$ for maximum value at (x', y', z') where $x' > x$, $y' > y$ and $z' > z$. The obtained value is incremented and stored at the point (x, y, z). We can update the value at (x, y, z) in 3-D Range tree and query for the maximum in $O(\log^3 n)$ time. For the $\mathcal{O}(\mathcal{R}^2)$ points it will take $\mathcal{O}(\mathcal{R}^2 \log^3 n)$ time to solve the MCL problem. In the deepest level of our range tree we are doing a 1-D range maximum query, with the query range always having the form $[x, n]$. According to Rahman *et. al.* such queries can be answered using a *Van Emde Boas* data structure in $\mathcal{O}(\log \log n)$ time [5]. Using this technique, the running time to solve the MCL (which in turn solves the LCPS problem) problem reduces to $O(\mathcal{R}^2 \log^2 n \log \log n)$.

5 Conclusion and Future Works

We have presented a $\mathcal{O}(n^4)$ time dynamic programming algorithm for solving the LCPS problem. Then, we have identified and studied some interesting relation of the problem with computational geometry. Then we solved the problem using a modified range tree data structure in $\mathcal{O}(\mathcal{R}^2 \log^2 n \log \log n)$ time. However, our results can be easily extended for the case where the two input strings are of different lengths. Further research can also be carried out towards studying different other variants of the LCPS problem.

References

1. Bentley, J.L., Friedman, J.H.: Data structures for range searching. ACM Comput. Surv. 11, 397–409 (December 1979)
2. Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel. Algorithmica 14, 355 – 366 (October 1995)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York
4. Hsu, P.H., Chen, K.Y., Chao, K.M.: Finding all approximate gapped palindromes. In: Proceedings of 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. pp. 1084 – 1093 (2009)
5. Iliopoulos, C., Rahman, M.: A new efficient algorithm for computing the longest common subsequence. Theory of Computing Systems 45, 355–371 (2009)
6. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theoretical Computer Science pp. 5365 – 5373 (November 2009)
7. Manacher, G.: A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. Journal of the ACM 22, 346 – 351 (July 1975)
8. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoretical Computer Science 410, 900–913 (March 2009)
9. Tomohiro, I., Shunsuke, I., Masayuki, T.: Palindrome pattern matching. In: Proceedings of 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. pp. 232 – 245 (2011)