Fundamenta Informaticae 129 (2014) 1–12
DOI 10.3233/FI-2014-860
IOS Press

Computing a Longest Common Palindromic Subsequence*

Shihabur Rahman Chowdhury, Md. Mahbubul Hasan,
Sumaiya Iqbal, M. Sohel Rahman†‡
AℓEDA group, Department of CSE,
Bangladesh University of Engineering & Technology
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 word which reads the same forward as it does backward. The *longest common palindromic subsequence (LCPS)* problem is a variant of the classic LCS problem which finds a 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 two novel algorithms to solve it. To the best of our knowledge, this is the first attempt to study and solve this 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. A *subsequence* of a string is obtained by deleting zero or more symbols of that string. A *common subsequence* of two strings is a

*Part of this research work was carried out under the research project titled "Next Generation Algorithms on Sequences" funded by Ministry of Education, Government of the People's Republic of Bangladesh.
†Partially supported by a Commonwealth Fellowship and an ACU Titular Fellowship
‡Address for correspondence: AℓEDA group, Department of CSE, Bangladesh University of Engineering & Technology, Dhaka - 1000, Bangladesh

subsequence common to both the strings. A *palindrome* is a word, phrase, number, or other sequence of units which reads the same forward as it does backward. The LCS problem for two strings is to find a common subsequence in both the strings, having maximum possible length. In the LCPS problem, the computed longest common subsequence, i.e., LCS, must also be a palindrome. More formally, given a pair of strings X and Y over an alphabet Σ, the goal of the LCPS problem is to compute a Longest Common Subsequence Z 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, 6, 8, 11, 13, 14, 15, 16]. The LCPS problem can be seen as a new addition to the already rich repertoire of problems related to palindromes. To the best of our knowledge, there exists no publication 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 [13]. Gusfield gave a linear-time algorithm to find all *maximal* palindromes in a string [7]. Porto and Barbosa gave an algorithm to find all *approximate* palindromes in a string [16]. Authors in [15] solved the problem of finding all palindromes in SLP (Straight Line Programs)-compressed strings. Additionally, a number of problems on variants of palindromes have also been investigated in the literature [8, 3, 11]. Very recently, I *et al.* worked on pattern matching problems involving palindromes [9].

Apart from being interesting from a pure theoretical point of view, the LCPS problem may turn out to be useful in computational biology as well. Biologists believe that palindromes play an important role in regulation of gene activity and other cell processes because these are often observed near promoters, introns and specific untranslated regions. Identifying palindromes could help in advancing the understanding of genomic instability [4], [12], [17]. Finding common palindromes in two gene sequences can be an important criterion to compare them, and also to find common relationships between them.

The rest of the paper is organized as follows. In Section 2 we give some definitions and introduce the notations used in the rest of the paper. We present a $\mathcal{O}(n^4)$ time Dynamic Programming algorithm to solve the *LCPS* problem in Section 3. In Section 4, we map the *LCPS* problem to a problem from computational geometry and present a $\mathcal{O}(\mathcal{R}^2 \log^2 n \log\log n)$ time algorithm to solve it. Finally, we conclude with some future directions of our work in Section 5.

2. Preliminaries

We assume a finite alphabet, Σ. Given a string $X = x_1x_2 \dots x_n$, $X_{i,j} = x_i \dots x_j$ $(1 \leq i \leq j \leq n)$ is a *substring* of X. A *palindrome* is a string which reads the same forward as it does backward. More formally, we say a string $Z = z_1z_2 \dots z_u$ is a palindrome *if and only if* $z_i = z_{u-i+1}$ for any $1 \leq i \leq \lceil \frac{u}{2} \rceil$. A *subsequence* of a string X is a sequence obtained by deleting zero or more characters from X. A subsequence Z of X is a *palindromic subsequence* if Z is a palindrome. 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 X and Y, having the maximum length is called a *Longest Common Palindromic Subsequence (LCPS)* and we denote it by $LCPS(X, Y)$.

[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.

For two strings $X = x_1x_2 \dots x_n$ and $Y = y_1y_2 \dots y_n$ we define a *match* to be an ordered pair (i, j) such that $x_i = y_j$. 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 $|\mathcal{M}| = \mathcal{R}$. We define, $\mathcal{M}_\sigma$ as a subset of $\mathcal{M}$ such that all matches within this set are due to a single character $\sigma \in \Sigma$. That is, $\mathcal{M}_\sigma = \{(i, j) : 1 \leq i \leq n, 1 \leq j \leq n \text{ and } x_i = y_j = \sigma \in \Sigma\}$, and $|\mathcal{M}_\sigma| = \mathcal{R}_\sigma$. Clearly, $\mathcal{M}_\sigma \subseteq \mathcal{M}$ and $\mathcal{M} = \bigcup_{\sigma \in \Sigma} \mathcal{M}_\sigma$. Each member of $\mathcal{M}_\sigma$ is called a *σ-match*.

3. A Dynamic Programming Algorithm

A brute-force approach to this problem would be to enumerate all the subsequences of X and Y and compare them, keeping track of the longest palindromic subsequence found. There are 2^n subsequences of any string of length n. So the brute force approach would lead to an exponential time algorithm. In this section, we will devise a dynamic programming algorithm for the LCPS problem. Here, we will see that the natural classes of subproblems for LCPS correspond to pairs of *substrings* of the two input sequences. We first present the following theorem which proves the optimal substructure property of the LCPS problem.

Theorem 3.1. Let X and Y be two strings of length n and $X_{i,j} = x_ix_{i+1} \dots x_{j-1}x_j$ and $Y_{k,\ell} = y_ky_{k+1} \dots y_{\ell-1}y_\ell$ are two substrings of them respectively. Let $Z = z_1z_2 \dots z_u$ 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 $x_i = x_j = y_k = y_\ell$ 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 $Y_{k+1,\ell}$).

Proof:
(1) By definition Z is a palindrome. Hence, we have $z_1 = z_u$. If $z_1 = z_u \neq a$ then we can append a at both ends of Z to obtain a common palindromic subsequence of $X_{i,j}$ and $Y_{k,\ell}$ of length $u + 2$, which contradicts the assumption that Z is an *LCPS* of $X_{i,j}$ and $Y_{k,\ell}$. So we must have $z_1 = z_u = a$. Now, the substring $z_2 \dots z_{u-1}$ with length $u - 2$ itself is a palindrome and it is common to both $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$. We need to show that it is an *LCPS*. For the purpose of contradiction let us assume that there is a common palindromic subsequence W of $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$ with length greater than $u - 2$. Then appending a to both ends of W will produce a common subsequence of $X_{i,j}$ and $Y_{k,\ell}$ with length greater than u, which is a contradiction.

(2) Since Z is a palindrome, $z_1 = z_u$. Since the condition $x_i = x_j = y_k = y_\ell$ does not hold, so z_1 and z_2 is not equal to at least one of x_i, x_j, y_k and y_ℓ. Therefore Z is a common palindromic subsequence of the substrings obtained by deleting at least one character from either end of $X_{i,j}$ or $Y_{k,\ell}$. If any pair of substrings obtained by deleting one character from either end of $X_{i,j}$ or $Y_{k,\ell}$ has a common palindromic subsequence W with length greater than u then it would also be a common palindromic subsequence of $X_{i,j}$ and $Y_{k,\ell}$, contradicting the assumption that Z is a *LCPS* of $X_{i,j}$ and $Y_{k,\ell}$.

This completes the proof. □

From Theorem 3.1, we see that if $x_i = x_j = y_k = y_\ell = a$ ($a \in \Sigma$), we must find an *LCPS* of $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$ and append a on its both ends to yield the *LCPS* of $X_{i,j}$ and $Y_{k,\ell}$. Otherwise, we must solve four subproblems and take the maximum of those. These four subproblems correspond to finding *LCPS* of:

(a) $X_{i+1,j}$ and $Y_{k,\ell}$ (b) $X_{i,j-1}$ and $Y_{k,\ell}$ (c) $X_{i,j}$ and $Y_{k,\ell-1}$ and (d) $X_{i,j}$ and $Y_{k+1,\ell}$

Let us define $lcps(i, j, k, \ell)$ to be the length of the *LCPS* of $X_{i,j}$ and $Y_{k,\ell}$. If either $i > j$ or $k > \ell$ then one of the substrings is empty and hence the length of our *LCPS* is 0. If both of the substrings has length 1, then the obtained *LCPS* will have length 1 if the single character substrings are equal. So we have the following two base cases,

$$lcps(i, j, k, \ell) = 0 \quad \text{if } i > j \text{ or } k > \ell \tag{1}$$

$$lcps(i, j, k, \ell) = 1 \quad \text{if } (i = j \text{ and } k = \ell) \text{ and } (x_i = x_j = y_k = y_\ell) \tag{2}$$

Using the base cases of Equations 1 and 2 and the optimal substructure property of LCPS (Theorem 3.1), we have the following recursive formula:

$$lcps(i, j, k, \ell) = \begin{cases} 0 & i > j \text{ or } k > \ell \\ 1 & (i = j \text{ and } k = \ell) \\ & \text{and} \\ & (x_i = x_j = y_k = y_\ell) \\ 2 + lcps(i+1, j-1, k+1, \ell-1) & (i < j \text{ and } k < \ell) \text{ and} \\ & x_i = x_j = y_k = y_\ell \\ \max(lcps(i+1, j, k, \ell), lcps(i, j-1, k, \ell), & (((i = j \text{ and } k \leq \ell) \text{ or} \\ lcps(i, j, k+1, \ell), lcps(i, j, k, \ell-1)) & (k = \ell \text{ and } i \leq j)) \text{ and} \\ & (x_i = x_j = y_k = y_\ell)) \\ & \text{or} \\ & ((i \leq j \text{ and } k \leq \ell) \text{ and} \\ & (x_i = x_j = y_k = y_\ell) \\ & \text{does not hold)} \end{cases} \tag{3}$$

The length of an *LCPS* between X and Y can be obtained by evaluating $lcps(1, n, 1, n)$. Since there are $\Theta(n^4)$ distinct subproblems, we can use dynamic programming to compute the solution in a bottom up manner. Algorithm 1 outlines the *LCPSLength* procedure which takes two strings X and Y as inputs. It stores the values of $lcps(i, j, k, \ell)$ in a $n \times n \times n \times n$ sized table named $lcps$. The table entries $i > j$, $k > \ell$ have value 0 since these entries correspond to at least one empty substring. We proceed in our computation with increasing length of the substrings. That is, table entries for substrings of length v are computed before that for substrings of length $v + 1$. The procedure returns the $lcps$ table and $lcps[1, n, 1, n]$ contains the length of an *LCPS* of X and Y. Theorem 3.2 gives us the running time of Algorithm 1.

Algorithm 1 LCPSLength(X,Y)

```
1: n ← length[X]
2: for i = 1 to n do
3:    for j = 1 to n do
4:       for k = 1 to n do
5:          for ℓ = 1 to n do
6:             if (i = j or k = ℓ) and (x_i = x_j = y_k = y_ℓ) then
7:                lcps[i, j, k, ℓ] = 1
8:             else
9:                lcps[i, j, k, ℓ] = 0
10:            end if
11:         end for
12:      end for
13:   end for
14: end for
15: for xLength = 2 to n do
16:    for yLength = 2 to n do
17:       for i = 1 to n − xLength + 1 do
18:          for k = 1 to n − yLength + 1 do
19:             j = i + xLength − 1
20:             ℓ = k + yLength − 1
21:             if x_i = x_j = y_k = y_ℓ then
22:                lcps[i, j, k, ℓ] = 2 + lcps[i + 1, j − 1, k + 1, ℓ − 1]
23:             else
24:                lcps[i, j, k, ℓ] = max(lcps[i+1, j, k, ℓ], lcps[i, j−1, k, ℓ], lcps[i, j, k+1, ℓ], lcps[i, j, k, ℓ−1])
25:             end if
26:          end for
27:       end for
28:    end for
29: end for
30: return lcps
```

Theorem 3.2. $LCPSLength(X, Y)$ computes the length of an LCPS of X and Y in $\mathcal{O}(n^4)$ time.

Proof:
The initialization step takes $\mathcal{O}(n^4)$ time. As the algorithm proceeds, it computes the *LCPS* of substrings of X and Y in such a way that substrings of length v is considered before substrings of length $v + 1$. Now, there are $\mathcal{O}(n^2)$ possible pairs of lengths between X and Y. For each of these pairs there are $\mathcal{O}(n^2)$ possible start position pairs. So the four nested loops in Lines 15 - 18 require $\mathcal{O}(n^4)$ time. And each table entry takes $\mathcal{O}(1)$ time to compute. So the table computation takes $\mathcal{O}(n^4)$ time in total. □

We can use the lengths computed in $lcps$ table returned by ***LCPSLength(X,Y)*** to construct an *LCPS* of X and Y. We begin at $lcps[1, n, 1, n]$ and trace back through the table. As soon as we find that $x_i = x_j = y_k = y_\ell$, we find an element of LCPS, and recursively try to find the *LCPS* for $X_{i+1,j-1}$ and $Y_{k+1,\ell-1}$. Otherwise, we find the maximum value in the $lcps$ table for $(X_{i+1,j}, Y_{k,\ell})$, $(X_{i,j-1}, Y_{k,\ell})$, $(X_{i,j}, Y_{k+1,\ell})$, $(X_{i,j}, Y_{k,\ell-1})$ and then use that value to compute subsequent members of LCPS recursively. Since at least one of i, j, k, ℓ is decremented in each recursive call, this procedure takes $\mathcal{O}(n)$ time to construct

an *LCPS* of X and Y. If the strings are of different length (say, $|X| = m, |Y| = n$) then there will be $\mathcal{O}(m^2n^2)$ pairs of substrings between X and Y. This results in a running time of $\mathcal{O}(m^2n^2)$.

4. A Second Approach

In this section, we present a second approach to efficiently solve the LCPS problem. In particular, we will first reduce our problem to a problem from computational geometry and then solve it with the help of a modified version of a range tree data structure. The resulting algorithm will run in $O(\mathcal{R}^2 \log^2 n \log\log n)$ time. Recall that, $\mathcal{R}$ is the number of ordered pairs at which the two strings match. First we make the following claim.

Claim 4.1. 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 ($\sigma \in \Sigma$).

Proof:
Since Z is a palindrome, we have, $z_i = z_{u-i+1}$ for $1 \leq i \leq \lceil \frac{u}{2} \rceil$. Since Z is common to both X and Y, each z_i , $1 \leq i \leq u$ corresponds to a σ-match between X and Y. Therefore, z_i and z_{u-i+1} constitute a σ-match pair. For odd length palindromes, there exists exactly one case when we have, $z_i = z_{u-i+1}$ and $i = u - i + 1$. This corresponds to the substrings of X and Y having exactly a single character, and the σ-match pair corresponds to a pair of same matches. So, z_i and z_{u-i+1} also forms a σ-match pair in this case. Now we can obtain σ-match pairs by pairing up each z_i and z_{u-i+1} for all $1 \leq i \leq \lceil \frac{u}{2} \rceil$. So we have decomposed Z into a set of σ-match pairs. □

It follows from Claim 4.1 that constructing a common palindromic subsequence of two strings can be seen as constructing an appropriate set of σ-match pairs between the input strings. An arbitrary pair of σ-matches, $\langle (i,k),(j,\ell) \rangle$ (say m_1), from among all pair of σ-matches between a pair of strings, can be seen as inducing a substrings pair in the input strings. Now suppose we want to construct a common palindromic subsequence Z with length u with m_1 at the two ends of Z. Clearly we have $z_1 = z_u = x_i = x_j = y_k = y_\ell$. Then to compute Z, we will have to recursively select σ-match pairs between the induced substrings $X_{i,j}$ and $Y_{k,\ell}$. In this way we shall get a set of σ-matches which will correspond to the common palindromic subsequences of the input strings. If we consider all possible σ-match pairs as the two end points of the common palindromic subsequence then the longest obtained one among all these will be an *LCPS* of the input strings. This is the basic idea for constructing *LCPS* in our new approach.

To compute $\mathcal{M}_\sigma$ for any $\sigma \in \Sigma$, we first linearly scan X and Y to compute two arrays, X_σ and Y_σ, which contain the indexes in X and Y where σ occurs. Then we take each pair between the two arrays to get all the ordered pairs where σ occurs in both strings.

4.1. Mapping the LCPS Problem to a Geometry Problem

Each match between the strings X and Y can be visualized as a point on a $n \times n$ rectangular grid where all the coordinates have integer values. Then, any rectangle in the grid corresponds to a pair of substrings of X and Y. 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 pair of σ-matches as the two ends of the common palindromic

subsequence and recursively construct the set of pair of σ-matches from within the induced substrings. Clearly, the rectangle induced by a pair of σ-matches will in turn contain some points (i.e matches) within it. We recursively continue within the induced sub-rectangles to find the *LCPS* between the substrings induced by the rectangles. When the recursion unfolds, we append the σ-match pair on the obtained sequence to get the *LCPS* that can be obtained with our σ-match pair corresponding to the two ends. Clearly, if we do this procedure for all such possible σ-match pairs then the longest of them will be our desired *LCPS* between the two strings. The terminating condition of this recursive procedure would be:

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 itself, it corresponds to the single character case in our Dynamic Programming solution.

So, in summary we do the following.

1. Identify an induced rectangle (say Ψ_1) by a pair of σ-matches.
2. Pair up σ-matches within Ψ_1 to obtain another rectangle and so on until we encounter either of the two terminating conditions T1 or T2.
3. We repeat the above for all possible σ-match pairs ($\forall\sigma \in \Sigma$).
4. At this point, we have a set of nested rectangle structures.
5. Here, 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 our 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.*

In what follows, we will refer to this problem as the Maximum Depth Nesting Rectangle Structures (MDNRS) problem.

4.2. A Solution to the MDNRS Problem

A σ-match pair, $\langle(i,k),(j,\ell)\rangle$ basically represents a 2-dimensional rectangle (say Ψ). Assume, without the loss of generality that (i,k) and (j,ℓ) correspond to the lower left corner and upper right corner of Ψ, respectively. In what follows, depending on the context, we will sometime use $\langle(i,k),(j,\ell)\rangle$ to denote the corresponding rectangle. Now, a rectangle $\Psi'(\langle(i',k'),(j',\ell')\rangle)$ will be nested within rectangle $\Psi(\langle(i,k),(j,l)\rangle)$ *if and only if* the following condition holds:

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

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

Now we convert a 2-dimensional rectangle $\Psi(\langle(i,k),(j,\ell)\rangle)$ to a 4-dimensional point $P_\Psi(i,k,-j,-\ell)$. We say that a point (x,y,z,w) is chained to another point (x',y',z',w') *if and only if* $(x,y,z,w) > (x',y',z',w')$. Then, it is easy to see that, a rectangle $\Psi'(\langle(i',k'),(j',\ell')\rangle)$, is nested within a rectangle $\Psi(\langle(i,k),(j,\ell)\rangle)$ *if and only if* 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-dimension having the maximum chain length. In what follows, we will refer to this problem as the Maximum Chain Length (MCL) Problem.

To solve this problem we will use a modified form of Range Tree data structure described in [1]. A range tree can store a set of points (or values in 1-dimension), and it can be used to answer queries like finding all the points (or values) within a given range, finding number of points (or values) in a given range etc. For our problem, we need to perform a range maximum query. We begin with a short description of 1-D range tree which can perform query to find out the maximum value in a given query range and later we describe a way to generalize it into higher dimensions.

1-D range tree for an N size array is a balanced binary search tree. Root of any subtree of the tree contains a range as a *key* and any *value* associated with that *key*. The range contained at the root node is $[1,n]$. Each range $[i,j]$ is then split into two halves i to $\left\lfloor \frac{i+j}{2} \right\rfloor$ (assigned to the root of left subtree of the node containing the key $[i,j]$) and $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$ to j (assigned to the right subtree of the node containing the key $[i,j]$). Each half is then processed similarly and split recursively until there is only one element left in the range. For example, for $N = 4$ there will be only 1 range in level 0, $[1,4]$. There will be 2 ranges in Level 1 namely, $[1,2]$ and $[3,4]$. In Level 2, there will be 4 ranges namely, $[1,1]$, $[2,2]$, $[3,3]$ and $[4,4]$.

We can perform `update(i,x)` and `findmax(i,j)` operations on a 1-D range tree constructed from an array $A[1\ldots n]$. An update operation will update the value at the i-th position of the array to x and also updates the corresponding range tree. The `findmax(i,j)` operation finds out the maximum value from index i to j (inclusive) of the array using the range tree. Both of these operations require $O(\log N)$ time [1].

In our problem, we require a data structure which can perform operation over a 3-D array. We want to update a value at Position (i,j,k) of the array and we want to perform a range maximum query in a cubic range. To be more specific, given (i,j,k) we want to perform a range maximum query in the range $[i,n]\times[j,n]\times[k,n]$, to find the maximum value at some index (i',j',k') where $i' > i$, and $j' > j$ and $k' > k$. This can be easily performed with the help of the multidimensional version of the range tree (Multi-level Range Trees). A d-dimensional range tree can be defined as a multi-level tree using an inductive definition on d. In the d-dimension, we shall store the point $(x_1, x_2, \ldots x_{d-1})$ in the tree, $\mathcal{T}$ with respect to the x_d-coordinates. For all nodes u of $\mathcal{T}$, we associate a $(d-1)$-dimensional multi-level range tree with respect to $(x_1, x_2, \ldots x_{d-1})$. During update and query operations for d-dimensional points we also perform the same operation recursively in the $(d-1)$-dimensional trees. By induction on d it can be trivially shown that any update and query operation in this tree can be done in $\mathcal{O}(\log^d n)$ time. So in 3-D, our query and update take $O(\log^3 n)$ time, where the array is of $n \times n \times n$ size.

Now we present a solution to the MCL problem for 2-D. Later we shall extend this solution for 4-D. In 2-D our points will be in the form of (x,y). We maintain a 1-D range tree, $\mathcal{T}$, over the range $[1\ldots n]$. The value stored at a node in the tree is the length of the longest chain that can be formed starting from any point with the x coordinate that falls within the node's range. Initially, all the values in

$\mathcal{T}$ are zero. We process the points in a non-increasing order of their y coordinates and in case of a tie, in non-decreasing order of their x coordinates. For each point (x, y), we perform a range maximum query over the range $[x, n]$ in $\mathcal{T}$ for the maximum value at index x', where $x' > x$. If the maximum value is K then we can construct a chain of length $K+1$ starting from the point (x, y), and it's immediate successor will be a point with x' as its abscissa. Now, we perform an update operation on the tree to update the value at index x with corresponding value $K+1$. More specifically, the update operation will update the values at nodes whose ranges contain x. Since $\mathcal{T}$ is balanced, any update or query operation can be done in $\mathcal{O}(\log n)$ time. The maximum value in $\mathcal{T}$ is the maximum length of the chain. If we also store at x the point (x, y), which yields the maximum chain length then we can use that to trace the chain later in linear time.

Now we can extend the solution of 2-D MCL Problem easily to 4-D using a 3-D range tree. We process the points (x, y, z, w) in non-increasing order of the highest dimension w. For each point (x, y, z, w) we perform a range query in $\mathcal{T}$ over the range $[x, n] \times [y, n] \times [z, n]$ for maximum value at (x', y', z') where, $x' > x$ and $y' > y$ and $z' > z$. The rest of the solution process is same as that of the 2-D solution. We can update the value at (x, y, z) in a 3-D range tree and perform a range maximum query in $O(\log^3 n)$ time. Since there are as many as $\mathcal{O}(R^2)$ points in the worst case, we can solve the LCPS problem in $O(\mathcal{R}^2 \log^3 n)$ time.

This running time can be further improved by modifying the range tree data structure. In the deepest level of our range tree we are performing a 1-D range maximum query. But our query range always has the form $[x, n]$. The 1-D range tree in the deepest level is used to answer such queries. Rahman *et al.* devised a method of answering range maximum/minimum query of the form $[1, x]$ in $\mathcal{O}(\log \log N)$ time [10] using the *Van Emde Boas* (vEB) tree [5] data structure, where the values stored in the tree are in the range $[1, N]$. Their update mechanism also required $\mathcal{O}(\log \log N)$ time to complete. The vEB tree supports insertion and deletion operations in $\mathcal{O}(\log \log N)$ time [5]. Now, a $[x, n]$ range query in a list is equivalent to a $[1, x]$ query in the list's reversal. Therefore, a $[x, n]$ query on a list can be performed using the same method given in [10]. So, instead of keeping a 1-D range tree in the deepest levels, we keep a vEB tree instead and perform the range queries in a way similar to the one described in [10]. Since our co-ordinate values are in the range $[1, n]$, therefore this will reduce both the query and update time in the 3-D range tree to $\mathcal{O}(\log^2 n \log \log n)$. Therefore, the running time to solve the LCPS problem reduces to $O(\mathcal{R}^2 \log^2 n \log \log n)$.

Algorithm 2 outlines the *LCPS-New* procedure which takes as input two strings X and Y, each of length n and the alphabet, Σ. The following theorem gives the worst case running time of the *LCPS-New* procedure.

Theorem 4.2. The LCPS-New procedure computes an LCPS of strings X and Y in $\mathcal{O}(\mathcal{R}^2 \log^2 n \log \log n)$ time.

Proof:
Since there are $\mathcal{R}$ matches between X and Y, we have $\mathcal{O}(\mathcal{R}^2)$ rectangles. Therefore, there are $\mathcal{O}(\mathcal{R}^2)$ points in 4-dimension. Since, $\mathcal{R} = \mathcal{O}(n^2)$ in the worst case, sorting the points require $\mathcal{O}(\mathcal{R}^2 \log \mathcal{R}^2) = \mathcal{O}(\mathcal{R}^2 \log n)$ time. Since the coordinate values are bounded within the range 1 to n, we can sort them in linear time using the counting sort algorithm. So this will reduce the sorting time to $\mathcal{O}(\mathcal{R}^2)$.

Construction of a 3-D range tree with $\mathcal{O}(\mathcal{R}^2)$ points take $\mathcal{O}(\mathcal{R}^2 \log^2 \mathcal{R}^2) = \mathcal{O}(\mathcal{R}^2 \log^2 n)$ time [1]. In our case, during the construction of the 3-D range tree a total of $\mathcal{O}(\mathcal{R}^2)$ insertions will be performed in

Algorithm 2 LCPS-New(X,Y,Σ)

for each $\sigma \in \Sigma$ **do**
  $\mathcal{M}_\sigma \leftarrow \phi$
  $X\sigma \leftarrow \phi$
  $Y\sigma \leftarrow \phi$
  for $i = 1$ to n **do**
    if $X[i] = \sigma$ **then**
      $X\sigma \leftarrow X\sigma \cup \{i\}$
    end if
    if $Y[i] = \sigma$ **then**
      $Y\sigma \leftarrow Y\sigma \cup \{i\}$
    end if
  end for
  for $i = 1$ to $|X\sigma|$ **do**
    for $j = 1$ to $|Y\sigma|$ **do**
      $\mathcal{M}_\sigma \leftarrow \mathcal{M}_\sigma \cup \{(X\sigma[i], Y\sigma[i])\}$
    end for
  end for
end for
$Rectangles \leftarrow \phi$ {$Rectangles$ contains the set of all rectangles}
for each $\sigma \in \Sigma$ **do**
  for each match $(i, k) \in \mathcal{M}_\sigma$ **do**
    for each match $(j, \ell) \in \mathcal{M}_\sigma$ **do**
      if $i \leq j$ and $k \leq \ell$ **then**
        $Rectangles \leftarrow Rectangles \cup \{(i, k), (j, \ell)\}$
      end if
    end for
  end for
end for
$P \leftarrow \phi$
for each $\Psi(i, k, j, \ell) \in Rectangles$ **do**
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(i, k, -j, -l)\}$
end for
Sort the points in $\mathcal{P}$ in non increasing order of 4th dimension
Initialize the multi-level range tree $\mathcal{T}$ with value zero in all the nodes
for each point $p(x, y, z, w) \in \mathcal{P}$ **do**
  Find the point (x', y', z') with maximum value in $\mathcal{T}$ such that $x' > x$ and $y' > y$ and $z' > z$.
  $K \leftarrow$ the value stored at (x', y', z')
  Update the value of (x, y, z) with $K + 1$
  Also store (x', y', z') in $\mathcal{T}$ at the node (x, y, z) as its successor.
end for
$lcps \leftarrow$ maximum value stored in $\mathcal{T}$
$LCPS \leftarrow$ trace the successors to obtain the sequence
return $LCPS$

the vEB trees at the deepest level, requiring a total of $\mathcal{O}(\mathcal{R}^2 \log\log n)$ time. Therefore, the construction of our 3-D range tree will require $\mathcal{O}(\mathcal{R}^2 \log^2 n + \mathcal{R}^2 \log\log n) = \mathcal{O}(\mathcal{R}^2 \log^2 n)$ time in total.

Each update and range query in the tree can be performed in $\mathcal{O}(\log^2 n \log\log n)$ time. Now, for $\mathcal{O}(\mathcal{R}^2)$ points, a total of $\mathcal{O}(\mathcal{R}^2)$ queries are made which takes a total of $\mathcal{O}(R^2 \log^2 n \log\log n)$ time. Therefore, the overall running time of our algorithm is $\mathcal{O}(\mathcal{R}^2 \log^2 n \log\log n + \mathcal{R}^2 \log^2 n) = \mathcal{O}(\mathcal{R}^2 \log^2 n \log\log n)$. □

The worst case running time of our algorithm becomes $\mathcal{O}(n^4 \log^2 n \log\log n)$, since $\mathcal{R} = \mathcal{O}(n^2)$. This is clearly worse than that of the Dynamic Programming algorithm ($\mathcal{O}(n^4)$). But in cases where we have $\mathcal{R} = \mathcal{O}(n)$, it exhibits very good performance. In such a case the running time reduces to $\mathcal{O}(n^2 \log^2 n \log\log n)$. Even for $\mathcal{R} = \mathcal{O}(n^{1.5})$, this algorithm performs better ($\mathcal{O}(n^3 \log^2 n \log\log n)$) than the DP algorithm.

5. Conclusion and Future Works

In this paper, we have introduced and studied the longest common palindromic subsequence (LCPS) problem, which is a variant of the classic LCS problem. We have first presented a dynamic programming algorithm to solve it, which runs in $\mathcal{O}(n^4)$ time. Then, we have identified and studied some interesting relation of the problem with a problem in computational geometry and devised an $\mathcal{O}(\mathcal{R}^2 \log^2 n \log\log n)$ time algorithm. In our results, we have assumed that the two input strings are of equal length n. However, our results can be easily extended for the case where the two input strings are of different lengths. To the best of our knowledge this is the first attempt in the literature to solve this problem.

As a future work we are interested in working on a variant of the LCPS problem, where the gap between two consecutive characters in the LCPS lies within some specified upper and lower bound in at least one of the input strings. It would be also interesting to study a more restrictive form of this problem, where the bounds on the gap is imposed for both of the input strings. Additionally, the two new computational geometry problems (MDNRS and MCL) introduced here seem to be interesting on their own right and could be worth further investigation.

References

[1] Bentley, J. L., Friedman, J. H.: Data Structures for Range Searching, *ACM Comput. Surv.*, **11**, December 1979, 397–409, ISSN 0360-0300.

[2] Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel, *Algorithmica*, **14**, October 1995, 355 – 366.

[3] Chen, K.-Y., Hsu, P.-H., Chao, K.-M.: Identifying Approximate Palindromes in Run-Length Encoded Strings, *Proceedings of 21st International Symposium, ISAAC 2010, Jeju, Korea, December 15-17, 2010*, 2010.

[4] Choi, C.: DNA palindromes found in cancer, *Genome Biology*, **6**, 2005, ISSN 1465-6906.

[5] van Emde Boas, P.: Preserving order in a forest in less than logarithmic time, *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, 1975.

[6] Galil, Z.: Real-time algorithms for string-matching and palindrome recognition, *Proceedings of the eighth annual ACM symposium on Theory of computing*, 1976.

[7] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, 1997.

[8] Hsu, P.-H., Chen, K.-Y., Chao, K.-M.: Finding All Approximate Gapped Palindromes, *Proceedings of 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009.*, 2009.

[9] I, T., Shunsuke, I., Masayuki, T.: Palindrome Pattern Matching, *Proceedings of 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011.*, 2011.

[10] Iliopoulos, C., Rahman, M.: A New Efficient Algorithm for Computing the Longest Common Subsequence, *Theory of Computing Systems*, **45**, 2009, 355–371, ISSN 1432-4350.

[11] Kolpakov, R., Kucherov, G.: Searching for gapped palindromes, *Theoretical Computer Science*, November 2009, 5365 – 5373.

[12] Lange, J., Skaletsky, H., van Daalen, S. K. M., Embry, S. L., Korver, C. M., Brown, L. G., Oates, R. D., Silber, S., Repping, S., Page, D. C.: Isodicentric Y Chromosomes and Sex Disorders as Byproducts of Homologous Recombination that Maintains Palindromes, *Cell*, **138**, September 2009, 855–869.

[13] Manacher, G.: A new Linear-Time On-Line Algorithm for Finding the Smallest Initial Palindrome of a String, *Journal of the ACM*, **22**, July 1975, 346 – 351.

[14] Martnek, T., Lexa, M.: Hardware acceleration of approximate palindromes searching, *Proceedings of The International Conference on Field-Programmable Technology*, 2008.

[15] 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**, March 2009, 900–913.

[16] Porto, A. H. L., Barbosa, V. C.: Finding Approximate Palindromes in Strings, *Pattern Recognition*, 2002.

[17] Tanaka, H., Bergstrom, D. A., Yao, M.-C., Tapscott, S. J.: Large DNA palindromes as a common form of structural chromosome aberrations in human cancers, *Human Cell*, **19**(1), 2006, 17–23, ISSN 1749-0774.