Math.Comput.Sci.
DOI 10.1007/s11786-016-0288-7

Palindromic Subsequence Automata and Longest Common Palindromic Subsequence

Md. Mahbubul Hasan · A. S. M. Sohidull Islam · M. Sohel Rahman · Ayon Sen

Received: 21 April 2014 / Revised: 28 June 2014 / Accepted: 13 November 2014

Abstract In this paper, we present a novel weighted finite automaton called palindromic subsequence automaton (PSA) that is a compact representation of all the palindromic subsequences of a string. Then we use PSA to solve the longest common palindromic subsequence problem. Our automata based algorithms are efficient both in theory and in practice.

Keywords Finite automata · Palindromes · Palindromic subsequences · Algorithms

Mathematics Subject Classification 68Q45 · 68W32

1 Introduction

A *string* is a sequence of symbols drawn from an alphabet Σ. A *subsequence* of a string is a sequence that can be derived by deleting zero or more symbols from it without changing the order of the remaining symbols. A *palindrome* is a string w such that $w = w^R$, where w^R is the reverse of w; often, w is said to be a *palindromic string*. For example, *ATTA* and *CATTAC* are palindromes. In the Palindromic Subsequence Problem, all the palindromic subsequences

Part of this research work was conducted when M. Sohel Rahman was on a Sabbatical Leave from BUET and was partially supported by a Commonwealth Fellowship.

Md. M. Hasan · A. S. M. S. Islam · M. S. Rahman (✉) · A. Sen
AℓEDA Group, Department of CSE, BUET, Dhaka 1000, Bangladesh
e-mail: msrahman@cse.buet.ac.bd

Md. M. Hasan
e-mail: mahbub86@cse.buet.ac.bd

Md. M. Hasan
Google, Zurich, Germany

A. S. M. S. Islam
School of Computational Science and Engineering, McMaster University, Hamilton, Canada
e-mail: sohansayed@cse.buet.ac.bd

A. Sen
Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, USA
e-mail: ayonsn@cse.buet.ac.bd

Published online: 02 February 2017

of a string are to be computed. A *common subsequence* of two strings is a subsequence common to both the strings. Additionally, if the common subsequence is a palindrome, it is called a *common palindromic subsequence*.

Stringology researchers have been conducting research on different problems related to palindromes on strings and sequences since long [1,7,10,19–21,23]. Palindromes appear frequently in DNA and are widespread in human cancer cells [24]. Notice that palindromes in biological context consider complement DNA characters in the second half. Identifying these parts of DNAs could aid in the understanding of genomic instability [3,25]. 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. So, finding palindromic subsequences in any genome sequence is important. Also finding common palindromes in two genome sequences can be an important criterion for comparing them, and also to find common relationships between them.

The problem of computing palindromes and variants in a single sequence has received much attention in the literature. An on-line sequential algorithm was given by Manacher [20] that finds all initial palindromes in a string. Another algorithm to find long approximate palindromes was given by Porto and Barbosa [23]. Gusfield gave a linear-time algorithm to find all maximal palindromes in a string [8]. Matsubara et al. in [21] solved the problem of finding all palindromes in SLP (Straight Line Programs)-compressed strings. Additionally, a number of variants of palindromes have also been investigated in the literature [2,10,19]. Very recently, I et al. worked on pattern matching problems involving palindromes [14]. Chuang, Lee and Huang [5] proposed an algorithm to solve the palindromic subsequence problem.

1.1 Our Contribution

In this paper we present a weighted finite automaton that is a compact representation of all the palindromic subsequences of a string. The space complexity of our approach is better than that of [5]. In particular, we need only $O(n^2)$ space to represent all the palindromic subsequences of the given string while the space complexity of [5] is directly proportional to the total number of palindromes which is exponential. Furthermore, as an interesting application of PSA, we show how we can solve the LCPS problem of two given strings. Our algorithm is an input sensitive algorithm with a time complexity of $O(\mathcal{R}_1\mathcal{R}_2|\Sigma|)$, where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the number of states of the involved automata. Notably, since both $\mathcal{R}_1$ and $\mathcal{R}_2$ can be $O(n^2)$, the time complexity of our algorithm is no better than $O(n^4|\Sigma|)$ in the worst case. However, for cases when $\mathcal{R}_1$ (and/or $\mathcal{R}_2$) becomes $o(n^2)$ we get a far better running time.

1.2 Roadmap

In Sect. 2, we present the Palindromic Subsequence Automaton. In Sect. 3, we present another automaton to find the LCPS of two strings which is derived from PSA presented in Sect. 2. In Sect. 4, we present extensive experimental results. Finally we briefly conclude in Sect. 5.

2 PSA: An Automaton to Represent All Palindromic Subsequences of a String

To design this automaton we need a string and its reverse. We will find the common subsequence automaton of these two strings. We will follow the techniques provided in [6,9]. The resulting automaton can be seen as a weighted version of the common subsequence automaton (CSA) between a string and its reverse.

Definition 1 [*Palindromic subsequence automata (PSA)*] Given a string S, let the reverse of S be S^R. A palindromic subsequence automaton (PSA) M accepts the first half of a palindromic subsequence of all palindromic subsequences of the given string S.

The PSA M is a 6 tuple $(Q, \sum, \delta, \sigma, q_0, F)$, where

- Q is a finite set of states. Here, Q is a subset of pairs of positions in S and S^R
- $\sum$ is the input alphabet

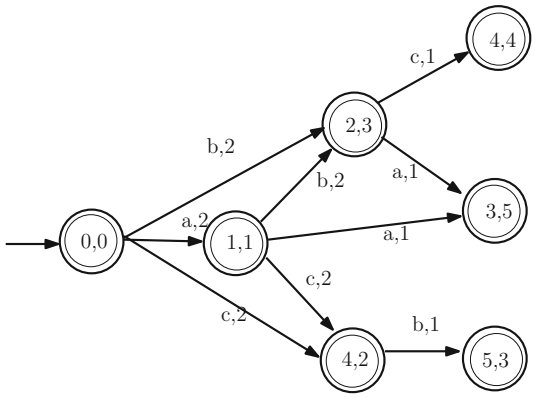

Fig. 1 A palindromic subsequence automata for string "abacbca"

- $\delta : Q \times \sum \longrightarrow Q$ is a transition function
- $\sigma : Q \times \sum \times Q \longrightarrow K$ assigns a edge cost between a pair of states. Here $K = \{0, 1, 2\}$. A state is said to be an 'error' state if the edge from or to it has cost 0.
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states. Here F consists of all the states of Q except the error states.

Here, Q and δ are defined in the same way as in [6,9]. Each state $q_a \in Q$ is associated with a pair of positions (a_i, a_j) where a_i and a_j refer to positions in S and S^R respectively and $S[a_i] = S^R[a_j]$. Let the length of S is n. So we have $1 \leqslant a_i, a_j \leqslant n$ for any state q_a. Now, for any state $q_b \in Q$ and $c \in \Sigma$ we have an edge from q_b to state q_a if and only if $S[a_i] = S^R[a_j] = c$, $b_i < a_i$, $b_j < a_j$ and there is no l, k such that $b_i < l < a_i$, $b_j < k < a_j$ and $S[l] = c$ or $S[k] = c$. Now we explain the definition of the function σ as follows. The edge cost could be one of 0, 1 and 2, i.e., $K = \{0, 1, 2\}$. The reason for such values will be clear from the following definition of the cost function:

$$\sigma(q_b, c, q_a) = \begin{cases} 2, & \text{if } a_i < n - a_j + 1. \\ 1, & \text{if } a_i = n - a_j + 1. \\ 0, & \text{if } a_i > n - a_j + 1. \end{cases} \tag{1}$$

A brief discussion on the elements of K, i.e., the possible values of edge cost is in order. We will construct a palindromic string based on accepted strings of PSA. If the accepted string is w, we find the reverse of w which is w^R and concatenate both to create a palindrome of even length. However for a palindrome having odd length we can not do that. In this case, we have to compute u such that $w = ua$ where $a \in \Sigma$. In this case, we can get a palindrome of the form uau^R. To determine whether an odd-length palindrome is there (in addition to the even-length one), we use the value two. The value of one is used to indicate that only an even-length palindrome of the form ww^R is there. A value of zero indicates that the corresponding character does not take part in the construction of a palindrome.

Example 1 Let, $S = abacbca$. The PSA of S is given in Fig. 1. Each state in the PSA is an ending state. In Fig. 1 each edge represents a transition. The character above each edge represents the transition character and the number above each edge represents the cost of it. Each state is represented by $[i, j]$, where i and j are indexes of S and S^R respectively. For example, for the edge between states [1, 1] and [2, 3], the transition character is b and the cost is 2. For simplicity of the figure, the 'error' states are omitted.

Now we discuss how PSA works. Assume that we have computed a PSA, M_1 of a given string S_1. Now we want to check whether a string $S = a_1 a_2 \ldots a_k$ is a palindromic subsequence of S_1. Intuitively, we can always obtain a longest palindromic subsequence of S_1 by first taking the longest common sequence (LCS) L of S_1 and S_1^R and then "reflecting" the first half of the result onto the second half; that is, if L has k characters, then we replace the last $\lfloor \frac{k}{2} \rfloor$ characters of L by the reverse of the first $\lfloor \frac{k}{2} \rfloor$ characters of L to obtain a longest palindromic subsequence of S_1. Obviously, this argument can be extended for any palindromic subsequence by taking any common subsequence of S_1 and

S_1^R. Now suppose that $S_h = a_1 a_2 \ldots a_{\lfloor \frac{k+1}{2} \rfloor}$. It follows from the discussion above that S will be a palindromic subsequence of S_1 if and only if S_h is a common subsequence of both S_1 and S_1^R. So, to decide whether S is a palindromic subsequence of S_1, it is sufficient to check whether S_h is accepted[1] by M_1. Hence, we have the following lemma.

Lemma 1 *Suppose we have constructed M_1 based on the string S_1. Further suppose that $S = a_1 a_2 \ldots a_k$ is a palindromic string and $S_h = a_1 a_2 \ldots a_{\lfloor \frac{k+1}{2} \rfloor}$. If S_h is accepted by M_1 then S is a palindromic subsequence of S_1.* □

2.1 Computing All Palindromic Subsequences

To find all the palindromic subsequences, all we need is to traverse M_1 and find the strings accepted by M_1 as follows. Suppose we get a string V_1 reaching a final state q_n. The state before that is q_{n-1} and the transition character is c. Let, $\sigma(q_{n-1}, c, q_n) = k_n$. Clearly, $k_n \in \{0, 1, 2\}$. If $k_n = 2$, we can get two palindromic subsequences. The first string is $V_1 V_1^R$. Suppose $V_1 = Uc$. Then, the second string is UcU^R. If $k_n = 1$ only one palindromic subsequence can be formed which is UcU^R. If $k_n = 0$, we can not get a palindromic subsequence with the current string reaching the final state q_n.

The following example explains the characteristics of PSA.

Example 2 In Fig. 1 a path from the starting node to [2, 3] is [0, 0], [1, 1], [2, 3]. In this case $k_n = 2$, i.e., the transition from [1, 1] to [2, 3] is 2. So, we can form two palindromic subsequences: *abba* and *aba*. But the transition from [1, 1] to [3, 5] has cost 1. A path from the starting node to [3, 5] is [0, 0], [1, 1], [3, 5]. As $k_n = 1$ only one palindromic subsequence, *aaa* can be formed.

The algorithm for constructing PSA is formally presented in Algorithm 1. The algorithm requires a simple pre-processing step which runs in $O(n)$ time. The input for the pre-processing is the string under consideration. And the out put is an array NP, which keeps track of the next position of a character of the string. To get the array NP we need another array SP. Both SP and NP are formally defined below.

Definition 2 Given a string S of length n on an alphabet Σ, the array SP is an array of integers of size $|\Sigma|$ such that for all $k \in \Sigma$, $SP[k]$ stores the leftmost occurrence of character k in the string $S[(i+1) \ldots n]$ for any $1 \leq i \leq n$. For $i = n$, $SP[k] = null$ for all $k \in \Sigma$.

Definition 3 Given a string S of length n, the (next position) array NP is an array of integers of size n. For all $1 \leq i \leq n$, $NP[i]$ stores the smallest integer j where $i < j < n$ such that $S[i] = S[j]$. If $SP[i]$ is the rightmost occurrence in S, then, $NP[i] = null$.

We start with $i = n$ and at each step i is decreased by 1. Let $p = S[i]$. If $SP[p]$ is 'null' then we assign $NP[i] = null$. Otherwise (i.e., if $SP[p]$ is not 'null'), we assign $NP[i] = SP[p]$ before updating $SP[p] = i$. The process will continue until the starting position of the string is reached, i.e., $i = 1$. The array NP will be used for the function NEXT_ MATCH() in Algorithm 1. Suppose for a given string S we have computed the NP array. Now if $S[i] = a$ then $NP[i]$ denotes the position of next occurrence of a. An example of the constructing of NP is given in Example 3.

Example 3 Let us consider the string $S = abcbac$. So $n = 6$ and $|\Sigma| = 3$. Table 1 shows the steps of constructing the NP array. The iteration begins at $i = 6$ and ends at $i = 1$. The third column of Table 1 reports the values of $NP[i]$ for $1 \leq i \leq 6$. The fourth, fifth and sixth columns denote the values of $SP[a]$, $SP[b]$ and $SP[c]$ after the update at that iteration.

In Algorithm 1 we maintain a queue C of states. In each step we dequeue a state $[i_1, j_1]$ from C and try to find next states of the dequeued state. Here i_1 and j_1 denotes the positions of S_1 and $S_2 = S_1^R$. The initial state is [0,0]. For each character $s \in \Sigma$ we find the next occurrences i_2 and j_3 in S_1 and S_2. The function NEXT_ MATCH() returns these next positions. Now if $i_2 \leq n - j_3 + 1$ then we will create a new state $[i_2, n - j_3 + 1]$ and enqueue this state in C. We also need to find the edge cost between these two states. If $i_1 < n - j_1 + 1$, then the cost will be 2 otherwise the cost will be 1.

[1] Notably, in some sense one could say that PSA recognizes half of a palindrome.

Table 1 Construction of the array NP

i	$p = S[i]$	$NP[i]$	$SP[a]$	$SP[b]$	$SP[c]$
6	c	*null*	*null*	*null*	6
5	a	*null*	5	*null*	6
5	b	*null*	5	4	6
3	c	6	5	4	3
2	b	4	5	2	3
1	a	5	1	2	3

Algorithm 1 PSA Construction

input: S_1: input string, $S_2 : REVERSE(S_1), n : LENGTH(S_1)$
output: PSA $M = (Q, \sum, \delta, \sigma, q_0, F)$

begin
$q_0 \leftarrow [0, 0]$
$Q \leftarrow \{[0, 0]\}$
$F \leftarrow \{q_0\}$
$C \leftarrow NEW - QUEUE()$
$ENQUEUE(C, [0, 0])$
while not EMPTY(C) **do**
  $[i_1, j_1] \leftarrow DEQUEUE(C)$
  for all $s \in \Sigma$ **do**
    $i_2 \leftarrow NEXT\_MATCH(S_1, i_1, s)$
    $j_3 \leftarrow NEXT\_MATCH(S_2, j_1, s)$
    $j_2 \leftarrow n - j_3 + 1$
    if $i_2 \leqslant j_2$ **then**
      $\delta([i_1, j_1], s) \leftarrow [i_2, j_2]$
      if $[i_2, j_2] \notin Q$ **then**
        $ENQUEUE(C, [i_2, j_2])$
        $Q \leftarrow Q \cup [i_2, j_2]$
        $F \leftarrow Q \cup [i_2, j_2]$
      end if
      if $i_2 < j_2$ **then**
        $\sigma([i_1, j_1], s, [i_2, j_2]) \leftarrow 2$
      else
        $\sigma([i_1, j_1], s, [i_2, j_2]) \leftarrow 1$
      end if
    end if
  end for
end while
end

2.2 Analysis

The preprocessing step requires $O(n)$ time since it is a loop which iterates over the string and in each step constant time is needed. As the PSA is derived from subsequence automata of two strings [6,9], the running time for Algorithm 1 to compute a PSA of two strings is $O(n + \mathcal{R}|\Sigma|)$ [9], where $\mathcal{R}$ is the number of states and Σ is the set of characters. Clearly, $\mathcal{R}$ is less than the total number of matches between the two strings (each match does not always produce a valid state). In the worst case, $\mathcal{R} = O(n^2)$. Hence the worst case running time for Algorithm 1 is $O(n^2|\Sigma|)$.

3 An Application of PSA: Computing an LCPS

The *longest common subsequence* (LCS) problem for two strings is to find a common subsequence in both the strings, having the maximum possible length. In the *longest common palindromic subsequence* (LCPS) problem,

the computed longest common subsequence must also be a palindrome. In what follows, for the sake of convenience we will assume that the given two strings, X and Y, are of equal length, n. But our result can be easily extended to handle two strings of different length.

Despite a plethora of work on problems related to palindromes concerning a single sequence, to the best of our knowledge, there has not been any work on the LCPS problem until very recently, when Chowdhury et al. [4] introduced two algorithms to solve the LCPS problem with time complexity $O(n^4)$ and $O(R^2 \log^2 n \log \log n)$, respectively. Here, the set of all ordered pairs of matches between the two strings is denoted by $\mathcal{M}$ and $|\mathcal{M}| = R$. Readers are kindly noted regarding the subtle difference of the two parameters R and $\mathcal{R}$ (see Sect. 2.2 for a definition of the latter).

In this section we discuss how to use PSA to compute an LCPS of two given strings. Our idea is to compute an automata called common palindromic subsequence automata (CPSA) as defined below.

Definition 4 [*Common palindromic subsequence automata (CPSA)*] Given two strings, a common palindromic subsequence automata (CPSA) accepts all common palindromic subsequences of the given strings. In other words if M_1 and M_2 are the PSA of the given two strings, then the CPSA of them is the intersection of M_1 and M_2.

We use $\mathcal{L}(M)$ to denote the language accepted by the finite automaton M. Then, $\mathcal{L}(M)$ can seen as a set of words such that for each of these words there exists a sequence of transitions from the initial state to a final state. We use the MaxLen() function that given a language returns one of the largest words of that language. Now we borrowed the concept of a Max Length Automaton from [18] which is defined below.

Definition 5 Given an acyclic deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, the Max Length Automaton is a finite automaton $M_M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(M_M) = MaxLen(\mathcal{L}(M))$.

Now in the LCPS problem, given two strings S_1 and S_2 our task is to find the longest common palindromic subsequence (LCPS). Now we have the following lemma.

Lemma 2 *To find the longest common palindromic subsequence (LCPS) of two strings S_1 and S_2 we need to compute the Max Length Automaton of the intersection of the PSA of S_1 and S_2.*

Proof Since the PSA of S_1 and S_2 can generate all the palindromic sub-sequences of S_1 and S_2, their intersection automata can generate all the palindromic common sub-sequences of S_1 and S_2. So, the Max Length Automata will give us the LCPS of S_1 and S_2. □

So we have the following simple algorithm to compute an LCPS.

- Step 1: Compute the PSA M_1 of S_1
- Step 2: Compute the PSA M_2 of S_2
- Step 3: Compute the CPSA M_3 by intersecting M_1 and M_2
- Step 4: Compute the Max Length Automaton M_4 of M_3

Notably, in Step 3, we use the algorithm presented in [18,22] with a slight modification. Example 4 shows the construction of a CPSA. The algorithm for constructing CPSA is formally presented in Algorithm 2.

In this algorithm two automata M_1 and M_2 are given as input. We need to find the intersection M of these two automata. Each state of M is comprised by one of the states of M_1 and M_2. We maintain a queue of states C which keeps the states of M. If the first states of M_1 and M_2 are q_0^1 and q_0^2 respectively then the first state of M is $[q_0^1, q_0^2]$. So at first we put this state inside C. Now we run a loop on C. At each step we dequeue a state from C, finds the next possible states of it and enqueue them inside C. The loop continues until the queue is empty. Let in a step we dequeue a state $[q^1, q^2]$. For each character s in Σ we try to find the next states of q^1 and q^2 in M_1 and M_2 respectively. Let those states are p^1 and p^2 respectively. Then we get a new state $[p^1, p^2]$ for M. We also need the transition cost from $[q_0^1, q_0^2]$ to $[p^1, p^2]$. If the transition cost between q^1 to p^1 and q^2 to p^2 are the same then we assign the same transition cost. otherwise we assign 1 as the transition cost. Finally we check whether the newly created state is already in C or not. If it is not in C then we enqueue this state in C.

Algorithm 2 Algorithm for Construction of CPSA

input: PSA $M_1 = (Q^1, \Sigma^1, \delta^1, \sigma^1, q_0^1, F^1)$, $M_2 = (Q^2, \Sigma^2, \delta^2, \sigma^2, q_0^2, F^2)$
output: CPSA $M = (Q, \Sigma, \delta, \sigma, q_0, F)$, $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$

begin
$Q \leftarrow \{[q_0^1, q_0^2]\}$
$F \leftarrow \{q_0\}$
$C \leftarrow NEW - QUEUE()$
$ENQUEUE(C, [q_0^1, q_0^2])$
while not EMPTY(C) **do**
  $[q^1, q^2] \leftarrow DEQUEUE(C)$
  for all $s \in \Sigma$ **do**
    $[p^1, p^2] \leftarrow [\delta^1(q^1, s), \delta^2(q^2, s)]$
    $\delta([q^1, q^2], s) \leftarrow [p^1, p^2]$
    if $\sigma(q^1, s, p^1) = \sigma(q^2, s, p^2)$ **then**
      $\sigma([q^1, q^2], s, [p^1, p^2]) \leftarrow \sigma(q^1, s, p^1)$
    else
      $\sigma([q^1, q^2], s, [p^1, p^2]) \leftarrow 1$
    end if
    if $[p^1, p^2] \notin Q$ **then**
      $ENQUEUE(C, [p^1, p^2]$
      $Q \leftarrow Q \cup \{[p^1, p^2]\}$
      $F \leftarrow \{[p^1, p^2]\}$
    end if
  end for
end while
end

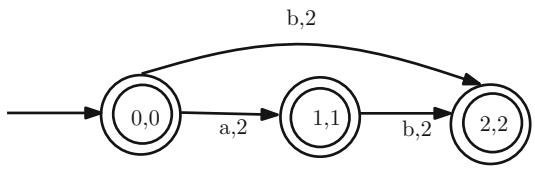

Fig. 2 A PSA for string "abba"

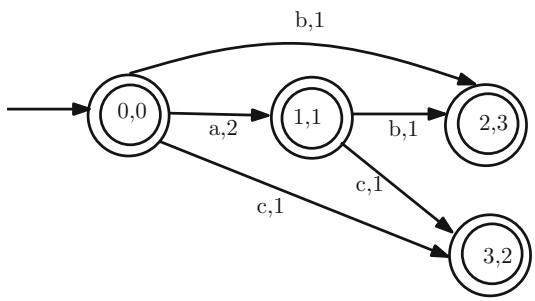

Fig. 3 A PSA for string "abca"

Example 4 Let $S_1 = abba$ and $S_2 = abca$ be two strings. Figures 2 and 3 represent the PSA's of S_1 and S_2 respectively. The CPSA is shown in Fig. 4. From that CPSA we can get four common palindromic subsequences of S_1 and S_2 namely a, b, aa and aba based on the paths and costs. The LCPS of S_1 and S_2 is aba.

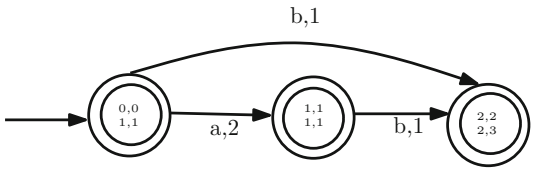

Fig. 4 A CPSA for string "abba" and "abca"

3.1 Analysis

Let the length of input strings be n and the total number of states in M_1 and M_2 be $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. Also let the alphabet is Σ. Now from the analysis of Algorithm 1 in Sect. 2.2 we know that the number of states in M_1 and M_2 are respectively $O(\mathcal{R}_1|\Sigma|)$ and $O(\mathcal{R}_2|\Sigma|)$. So, the time complexity of constructing M_3 (presented in Algorithm 2) will be $O(\mathcal{R}_1\mathcal{R}_2|\Sigma|)$ [18]. The time complexity of constructing M_4 will also be $O(\mathcal{R}_1\mathcal{R}_2|\Sigma|)$. We can traverse through one of the longest paths of M_4 and the accepting string will be an LCPS of the given two strings. As the length of LCPS is at most n so finding an LCPS in M_4 needs only $O(n)$ times. Thus the total complexity of computing an LCPS is $O(n + \mathcal{R}_1|\Sigma| + n + \mathcal{R}_2|\Sigma| + \mathcal{R}_1\mathcal{R}_2|\Sigma|)$. As $\mathcal{R}_1$ and $\mathcal{R}_2$ can be at most $O(n^2)$ each so worst case the running time is $O(n^4|\Sigma|)$

3.2 Comparison of the Algorithms

As has been mentioned above, to the best of our knowledge the only work in the literature that presents algorithm to compute LCPS is the very recent work of Chowdhury et al. [4]. In [4] two algorithms were provided. The first algorithm, referred to as CHIR-1 henceforth, runs in $O(n^4)$ time and the second one, referred to as CHIR-2 henceforth, runs in $O(R^2 \log^2 n \log\log n)$ time where R is the number of matches. In this section we present a theoretical comparison of the running time of our algorithm with CHIR-1 and CHIR-2.

Since $\mathcal{R}_1 = O(n^2)$ and $\mathcal{R}_2 = O(n^2)$, the running time of Algorithm 2 becomes $O(n^4|\Sigma|)$ in the worst case, which is not better than that CHIR-1 presented in [4]. But in cases where we have $\mathcal{R}_1 = O(n)$ and $\mathcal{R}_2 = O(n)$, it exhibits a very good performance. In such cases the running time reduces to $O(n^2|\Sigma|)$. Even for $\mathcal{R}_1 = \mathcal{R}_2 = O(n^{1.5})$ this algorithm performs better ($O(n^3|\Sigma|)$) than CHIR-1. Algorithm 2 also performs better than CHIR-2 in some of the cases. Clearly $R \geqslant \mathcal{R}_1$ and $R \geqslant \mathcal{R}_2$ and in most cases $|\Sigma| < \log^2 n \log\log n$. And if Σ is constant then our algorithm always outperforms CHIR-2.

It may be noted here that our algorithms to compute LCPS are input sensitive algorithms and exhibit better performance than the algorithms in the literature for certain special types of input. Such input sensitive algorithms have received significant attention in the literature especially for the problem of computing longest common subsequences and variants. To the best of our knowledge the first attempt to provide such an input sensitive algorithm dates back to 1977 when Hunt and Szymanski presented an $O((\mathcal{R}+n)\log n)$ algorithm for the classic LCS problem [13]. They also cited applications, where $\mathcal{R} \sim n$ and thereby claimed that for these applications the algorithm would run in $O(n \log n)$ time. Since then, a significant number of such input sensitive algorithms for the classic LCS problem and variants thereof have been published in the literature (e.g., [4,15–17]). To this end our algorithm can be seen as a new addition to this repertoire of algorithms. Finally, the Palindromic Subsequence Automaton seems to be of independent interest and may be found to be useful in different other problems in Stringology.

4 Experiments

In addition to a theoretical analysis presented, in this paper we have made an effort to experimentally analyse the performance of PSA and CPSA. Both the running time and space complexity of the algorithms presented here

Table 2 Number of states in PSA

n	$\|\Sigma\| = 4$	$\|\Sigma\| = 8$	$\|\Sigma\| = 12$	$\|\Sigma\| = 16$	$\|\Sigma\| = 20$	$\|\Sigma\| = 24$	$\|\Sigma\| = 28$
100	715	494	384	322	282	255	232
250	4582	3061	2275	1831	1546	1350	1198
500	18,588	12,148	8962	7111	5903	5083	4486
850	53,707	35,150	25,723	20,353	16,776	14,356	12,551
1000	74,401	48,645	35,478	28,008	23,167	19,720	17,261
1200	107,539	69,964	51,211	40,257	33,194	28,248	24,669
1500	167,600	109,669	79,886	62,637	51,792	43,929	38,378
5000	1,873,262	1,214,192	79,886	691,858	567,412	481,795	419,028
10,000	7,489,599	4,862,742	3,529,004	2,762,079	2,263,614	1,922,768	1,669,577

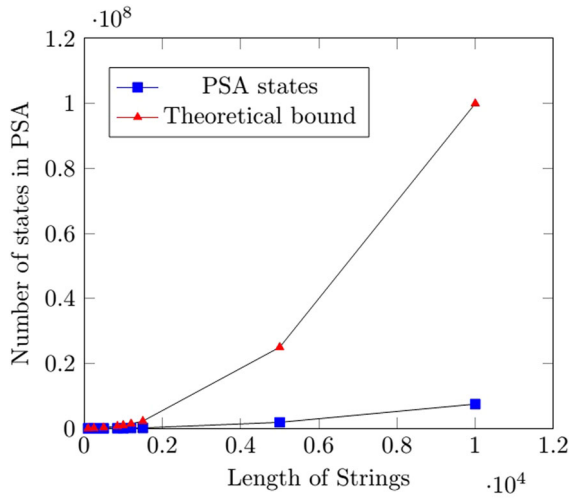

Fig. 5 Comparison among number of states of PSA with the theoretical upper bound

depend on the number of states of PSA and CPSA. So in our experiments we investigate how the number of states increases in practice with the increase of n and Σ for both PSA and CPSA. The experiments were conducted on a laptop with 8GB RAM, second Generation Core i7 Processor and 750GB of hard drive. The codes were written in Java. We have conducted experiments on both real data and randomly generated sequences. The complete source code and the data used are available at http://goo.gl/NNmQc6.

4.1 Random Data

For random Data, the strings have been generated randomly based on a given alphabet, Σ. For PSA the length of the string, n varies among 100, 250, 500, 850, 1000, 1200, 1500, 5000 and 10,000. The alphabet size $|\Sigma|$ varies among 4, 8, 12, 16, 20, 24 and 28. For each combination of n and $|\Sigma|$, we have generated 50 strings and have calculated (and reported) the average number of states of the PSA. These are presented in Table 2. It can be seen from Table 2 that by increasing n the number of states in PSA naturally increases. However this increase is not really as much as suggested by the theoretical analysis. For example when $n = 10{,}000$ and $|\Sigma| = 4$ the number of states of PSA should be in the range of 100,000,000 whereas in practice, it is only 7,489,599. Figure 5 compares the total number of states of PSA for different n and $|\Sigma|$ with the theoretical upper bound, i.e., n^2. Increasing $|\Sigma|$ decreases the number of states of PSA as is evident in Table 2 as well as in Fig. 6 which compares the total number of states where $|\Sigma|$ increases from 4 to 28 for $n = 10{,}000$.

Now, let us focus our attention to CPSA. As before our goal is to check how the number of states in CPSA changes with the change in n and $|\Sigma|$. While conducting the experiments the lengths of the two strings have been

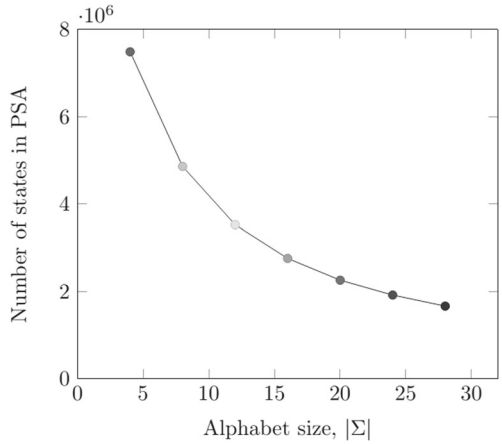

Fig. 6 Comparison among number of the states of PSA with different $|\Sigma|$ while $n = 10{,}000$

Table 3 Number of states in CPSA

n	$\|\Sigma\| = 4$	$\|\Sigma\| = 8$	$\|\Sigma\| = 12$	$\|\Sigma\| = 16$	$\|\Sigma\| = 20$	$\|\Sigma\| = 24$	$\|\Sigma\| = 28$
30	127	61	39	27	24	19	16
40	426	165	88	58	46	38	32
50	1027	385	199	125	92	70	58
62	2604	895	441	276	186	147	106
80	8525	2799	1304	720	482	334	245
100	24,634	7786	3299	1705	1122	788	564
120	58,311	17,239	7137	3786	2350	1574	1212
150	170,738	46,648	19,237	10,026	3957	4033	2782
170	277,262	84,245	33,298	17,043	9995	6463	4567
200	673,053	150,374	68,762	36,851	20,198	11,937	9730

kept equal. The length of the string, n varies among 30, 40, 50, 62, 80, 100, 120, 150, 170, 200. The alphabet size $|\Sigma|$ varies among 4, 8, 12, 16, 20, 24. For each combination of n and $|\Sigma|$ we have generated 50 strings and have calculated (and reported) the average number of states in the CPSA. The results are given in Table 3. As is evident from Table 3 the number of states in CPSA is far less than n^4, the theoretical worst case bound. It is even less than the square of the number of states in PSA (see Table 2). For example when $n = 200$ and $|\Sigma| = 4$, the number of states of CPSA should be in the range of 200000000 where in practice it is only 673053. So despite a theoretical worst case bound of $O(n^2)$ for PSA and $O(n^4)$ for CPSA, practically the states never reach that limit (and hence the time requirement as well is much low in practice). Figures 7 and 8 compares the total number of states of CPSA for different n and $|\Sigma|$ with the theoretical upper bound, i.e., n^4. Increasing $|\Sigma|$ decreases the number of states of CPSA as is evident in Table 3 as well as in Fig. 9 which compares the total number of states where $|\Sigma|$ increases from 4 to 28 for $n = 200$.

4.2 Real Data

We also have performed experiments on real data. Here we conduct two different sets of experiments. In the first setup we have chosen ACIN-1 as a sample dataset [11]. ACIN-1 stands for apoptotic chromatin condensation inducer 1 and it is a protein-coding gene. The length of the sequence is 2170. Since this is DNA sequence, the alphabet size is 4. For PSA the length of the string, n have been varied among 100, 250, 500, 850, 1000, 1200, 1500. For each length we have chosen 50 random substrings from ACIN-1. To get a random substring from ACIN-1 we generated a random number x less than the length of ACIN-1. Then for a length ℓ the random substring's starting position

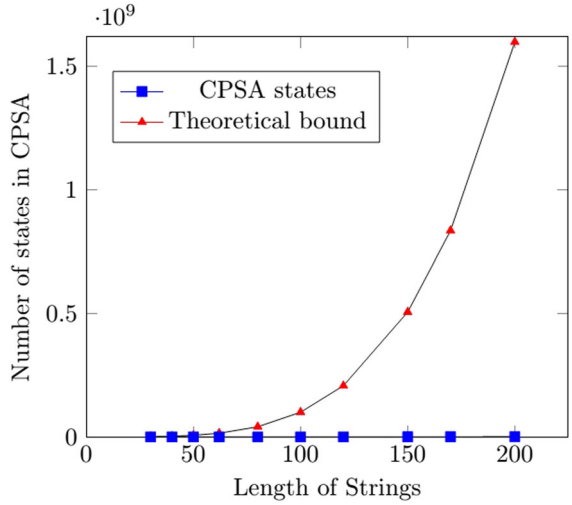

Fig. 7 Comparison among number of states of CPSA with the theoretical upper bound

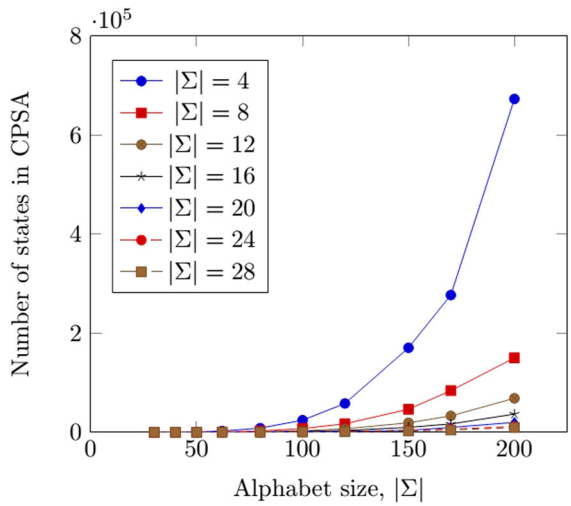

Fig. 8 Comparison among the number of the states of CPSA with different n and $|\Sigma|$

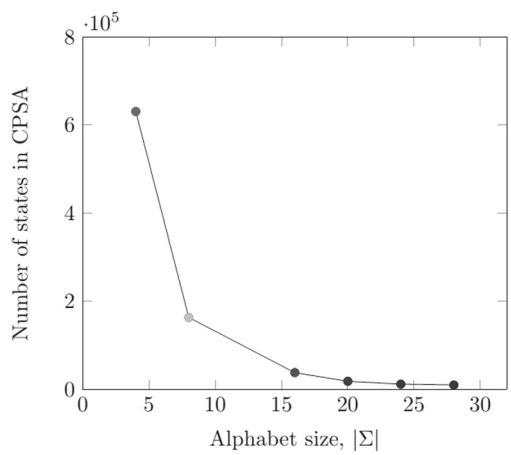

Fig. 9 Comparison among the number of the states of CPSA with different $|\Sigma|$ while $n = 200$

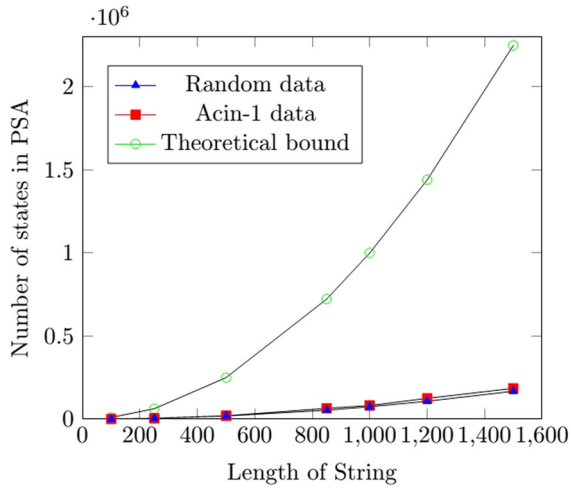

Fig. 10 Comparison among real data and random data for number of states of PSA

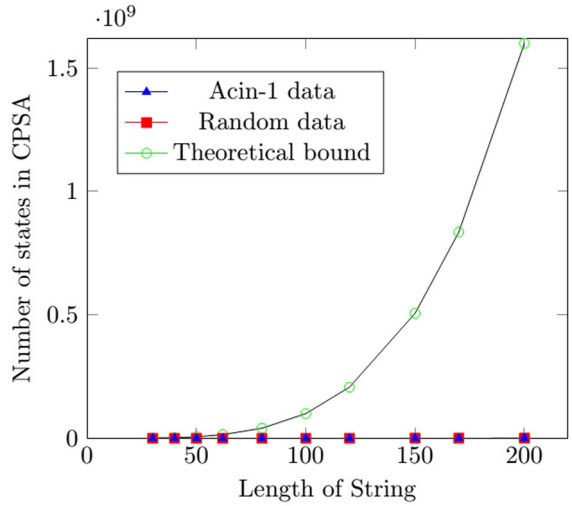

Fig. 11 Comparison among real data and random data for number of states of CPSA

will be x and ending position $x + \ell - 1$. We have calculated (and reported) the average number of states in the PSA for each length. Similarly for CPSA we have chosen the 50 random substrings for each length and calculated (and reported) the average number of states in CPSA. But in this case the length has been varied among 30, 40, 50, 62, 80, 100, 120, 150, 170, 200. Figures 10 and 11 compare the real data and random data for the number of states in PSA and CPSA respectively. From the figures it can be seen that number of states for both PSA and CPSA are nearly same for real and random data and both are significantly lower than the theoretical upper bound. Finally we perform experiments on some virus genomes. We use 30 virus genomes with length varying from 1378 to 10,035. The virus genomes were chosen from [12]. Table 4 shows the name, genome length and number of states of PSA constructed on the virus genomes. For computing the states of CPSA we have chosen five virus genomes, taken prefix of length 200 from each of them and computed CPSA between each possible pair of the prefixes. Table 5 shows the number of states in CPSA along with the virus pair. The number of states in CPSA is similar to the random data and is much lower than the theoretical bound.

Table 4 Number of states in PSA for virus genomes

Virus	Length, n	n^2	States in PSA
Anguilla anguilla circovirus	1378	1,898,884	153,973
Acartia tonsa copepod circovirus	1670	2,788,900	215,251
Barbel circovirus	1957	3,829,849	297,141
Beak and feather disease virus	1993	3,972,049	316,859
Avian gyrovirus2	2383	3,972,049	447,647
Ageratum yellow vein china virus	2739	7,502,121	617,824
Alternanthera yellow vein virus	2745	7,535,025	593,094
Ageratum enation virus	2746	7,540,516	576,621
Allamanda leaf curl virus	2755	7,590,025	598,499
Allamand leaf mottle distortion virus	2772	7,683,984	607,818
Bamini virus	2806	7,873,636	602,244
Axonopus compressus streak virus	2858	8,168,164	622,215
UR 2Sarcoma virus	3166	10,023,556	799,487
GroundS quirrel hepatitis virus	3311	10,962,721	876,096
Murine osteosarcoma virus	3811	14,523,721	1,153,241
Marine gokusho virus	4129	17,048,641	1,393,466
Snake parvovirus1	4432	19,642,624	1,553,385
Mouse parvovirus1	5144	26,460,736	2,195,204
Microvirus CA82	5514	30,404,196	2,454,311
Porcine parvovirus6	6148	37,797,904	2,981,385
Enterobacteria phageI2-2	6744	45,481,536	3,522,099
Ralstonia phage RSS0	7288	53,114,944	4,185,181
Vibrio phage Vf12	7965	63,441,225	4,776,207
Aconitum latent virus	8657	74,943,649	5,609,309
Caprine arthritis-encephalitis virus	9189	84,437,721	7,366,134
Human endogenous retrovirus K113	9472	89,718,784	7,437,458
Bidens mottle virus	9741	94,887,081	7,452,907
Acyrthosiphon pisum virus	10,035	100,701,225	8,352,788

Table 5 Number of states in CPSA for virus genomes

Virus pair	States in CPSA
Anguilla anguilla circovirus-acartia tonsa copepod circovirus	653,099
Anguilla anguilla circovirus-barbel circovirus	614,324
Anguilla anguilla circovirus-beak and feather disease virus	541,513
Anguilla anguilla circovirus-avian gyrovirus2	444,612
Acartia tonsa copepod circovirus-barbel circovirus	645,007
Acartia tonsa copepod circovirus-beak and feather disease virus	558,788
Acartia tonsa copepod circovirus-avian gyro virus2	518,258
Barbel circovirus-beak and feather disease virus	597,021
Barbel circovirus-avian gyrovirus2	454,043
Beak and feather disease virus-avian gyrovirus2	436,198

5 Conclusion

In this paper, we have introduced and presented PSA, a novel weighted finite automata that is a compact representation of all the palindromic subsequences of a string. The space complexity of our approach is better than that of [5]. Furthermore we use the PSA to solve the LCPS problem. In particular we present CPSA which is the intersection automata of two PSA of two given strings. The time complexity of our algorithm to solve the LCPS problem is $O(n+\mathcal{R}_1|\Sigma|+n+\mathcal{R}_2|\Sigma|+\mathcal{R}_1\mathcal{R}_2|\Sigma|)$ where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the number of states of the respective automaton. Our algorithm also performs better than the algorithms presented in [4] for $\mathcal{R}_1 = \mathcal{R}_2 = n$. We also present experimental results which suggest that both PSA and CPSA perform very well in practice.

References

1. Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel. Algorithmica **14**(4), 355–366 (1995)
2. Chen, K.-Y., Hsu, P.-H., Chao, K.-M.: Identifying approximate palindromes in run-length encoded strings. ISAAC **2**, 339–350 (2010)
3. Choi, C.Q.: DNA palindromes found in cancer. The Scientist (2005)
4. Chowdhury, S.R., Hasan, M.M., Iqbal, S., Rahman, M.S.: Computing a longest common palindromic subsequence. Fundam. Inform. **129**(4), 329–340 (2014)
5. Chuang, K., Lee, R., Huang, C.: Finding all palindrome subsequences in a string. In: The 24th Workshop on Combinatorial Mathematics and Computation Theory (2007)
6. Farhana, E., Rahman, M.S.: Doubly-constrained LCS and hybrid-constrained LCS problems revisited. Inf. Process. Lett. **112**(13), 562–565 (2012)
7. Galil, Z.: Real-time algorithms for string-matching and palindrome recognition. In: STOC, pp. 161–173 (1976)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
9. Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S.: Online construction of subsequence automata for multiple texts. In: SPIRE, pp. 146–152 (2000)
10. Hsu, P.-H., Chen, K.-Y., Chao, K.-M.: Finding all approximate gapped palindromes. In: ISAAC, pp. 1084–1093 (2009)
11. http://0-www.ncbi.nlm.nih.gov.ilsprod.lib.neu.edu/nuccore/acin02000001.1. Accessed 5 Jan 2014
12. http://www.ncbi.nlm.nih.gov/genomes/genomesgroup.cgi. Accessed 10 July 2014
13. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest subsequences. Commun. ACM **20**(5), 350–353 (1977)
14. Tomohiro, I., Inenaga, S., Takeda, M.: Palindrome pattern matching. In: CPM, pp. 232–245 (2011)
15. Iliopoulos, C.S., Rahman, M.S.: Algorithms for computing variants of the longest common subsequence problem. Theor. Comput. Sci. **395**(2–3), 255–267 (2008)
16. Iliopoulos, C.S., Rahman, M.S.: New efficient algorithms for the LCS and constrained LCS problems. Inf. Process. Lett. **106**(1), 13–18 (2008)
17. Iliopoulos, C.S., Rahman, M.S.: A new efficient algorithm for computing the longest common subsequence. Theory Comput. Syst. **45**(2), 355–371 (2009)
18. Iliopoulos, C.S., Rahman, M.S., Vorácek, M., Vagner, L.: Finite automata based algorithms on subsequences and supersequences of degenerate strings. J. Discrete Algorithms **8**(2), 117–130 (2010)
19. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theor. Comput. Sci. **410**(51), 5365–5373 (2009)
20. Manacher, G.K.: A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
21. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theor. Comput. Sci. **410**(8–10), 900–913 (2009)
22. Melichar, B., Holub, J., Muzatko, P.: Language and Translation. Publishing House of CTU (1997)
23. Porto, A.H.L., Barbosa, V.C.: Finding approximate palindromes in strings. Pattern Recognit. **35**(11), 2581–2591 (2002)
24. Tanaka, H., Bergstrom, D.A., Yao, M.-C., Tapscott, S.J.: Widespread and nonrandom distribution of dna palindromes in cancer cells provides a structural platform for subsequent gene amplification. Nat. Genet. **37**(3), 320–327 (2005)
25. Tanaka, H., Tapscott, S.J., Trask, B.J., Yao, M.C.: Short inverted repeats initiate gene amplification through the formation of a large DNA palindrome in mammalian cells. Natl. Acad. Sci. **99**(13), 8772–8777 (2002)