Palindrome Pattern Matching

Tomohiro I[1], Shunsuke Inenaga[2], and Masayuki Takeda[1]

[1]Department of Informatics, Kyushu University
[2]Graduate School of Information Science and Electrical Engineering,
Kyushu University
744 Motooka, Nishiku, Fukuoka 819–0395, Japan
{tomohiro.i, takeda}@inf.kyushu-u.ac.jp
inenaga@c.csce.kyushu-u.ac.jp

Abstract. A palindrome is a string that reads the same forward and backward. For a string x, let $\mathit{Pals}(x)$ be the set of all maximal palindromes of x, where each maximal palindrome in $\mathit{Pals}(x)$ is encoded by a pair (c, r) of its center c and its radius r. Given a text t of length n and a pattern p of length m, the palindrome pattern matching problem is to compute all positions i of t such that $\mathit{Pals}(p) = \mathit{Pals}(t[i : i + m - 1])$. We present linear-time algorithms to solve this problem.

1 Introduction

A palindrome is a symmetric string that reads the same forward and backward. Namely, a string w is a palindrome if $w = xax^R$ where x is a string, x^R is a reversal of x, and a is either a single character or the empty string.

Recently, palindromic structures in strings have been extensively studied: A string of length n is called *palindromic rich* (or simply *rich*) if it contains $n + 1$ distinct palindromes (including the empty string). It is known that any string of length n can contain at most $n + 1$ distinct palindromes [6]. A unified study of palindromic richness of finite and infinite strings was initiated in [7]. A close relationship between palindromic richness and the Burrows-Wheeler transform [5] was recently discovered in [16]. Another concept regarding palindromic structures is *palindrome complexity* [1, 4, 2] of a string, which is the number of palindromic substrings of a given length in the string.

There exist several efficient algorithms that solve interesting problems on palindromes: A linear-time algorithm to check if a given string is palindromic rich or not, is presented in [8]. One can compute the set of all maximal palindromes of a given string in linear time [13]. The reverse engineering problem of computing a string from a given set of maximal palindromes is solvable in linear time [11], and its closely related problem is also considered in [14].

In this paper, we introduce a new paradigm of pattern matching based on palindromes in strings. Two strings of same length m are said to be *pal-equivalent* iff the length of the maximal palindrome at every center in the strings is equal [11]. Given a text string t and a pattern string p, we are interested in finding all text positions i ($1 \leq i \leq n$) such that p and $t[i : i + m - 1]$ are pal-equivalent, where n and m are text and pattern lengths, respectively. This problem is called the *palindrome pattern matching.*

It is not difficult to see that the palindrome pattern matching problem can be solved in $O(nm)$ time: We pre-compute all maximal palindromes for t and p using linear time algorithms [13, 9]. For every text position i, we compare the length of the maximal palindromes of t at position $i + j - 1$ and that of p at position j for every $1 \leq j \leq m$. If a maximal palindrome of the text "goes over" the interval $[i : i+j-1]$, then the left and right arms of the maximal palindrome are trimmed accordingly for comparison.

There exists a linear-time algorithm for small alphabets. In [11] it was shown that if the alphabet size is at most 3, then two strings are pal-equivalent iff those strings parameterized match [3]. Hence the palindrome pattern matching can be solved in $O(n + m)$ time for ternary and smaller alphabets.

In this paper, we present efficient solutions for larger alphabets. Firstly, we present an algorithm which solves the problem in $O(n+m)$ time for *arbitrary* alphabets. This algorithm is a palindrome-pattern-matching version of the Morris-Pratt [15] pattern matching algorithm. Secondly, we propose another algorithm that uses a new text indexing structure called the *palindrome suffix trees.* We show that palindrome suffix trees can be constructed in $O(n \log \sigma)$ time, where σ is the alphabet size. Using the palindrome suffix tree, we can solve the problem in $O(m \log \sigma + r)$ time, where r is the number of text positions to report.

The algorithms of this paper are applicable to several practical problems, e.g., in bioinformatics. For instance, similar palindromic sequences often need to be identified in DNA and RNA sequence analysis [9]. Sequences having similar palindromic structures may code for similar 3-D structures of the respective molecules, leading to possible functional interpretation of the identified sequences. Due to the size of genomes, efficiency of search methods is of great importance.

2 Preliminaries

Let Σ be a finite *alphabet.* An element of Σ^* is called a *string.* The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i : j] = \varepsilon$ if $j < i$.

For any string w, let w^R denote the reversed string of w, that is, $w^R = w[|w|] \cdots w[2]w[1]$. A string w is called a *palindrome* if $w = w^R$. If $|w|$ is even, then w is called an *even palindrome*, that is, $w = xx^R$ for some $x \in \Sigma^*$. If $|w|$ is odd, then w is called an *odd palindrome*, that is, $w = xax^R$ for some $x \in \Sigma^*$ and $a \in \Sigma$. The *radius* of a palindrome w is $\frac{|w|}{2}$.

The *center* of a palindromic substring $w[i : j]$ of a string w is $\frac{i+j}{2}$. A palindromic substring $w[i : j]$ is called the *maximal palindrome* at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $w[i : j]$, i.e., if $w[i-1] \neq w[j+1]$, $i = 1$, or $j = |w|$. In particular, a maximal palindrome $w[i : |w|]$ is called a *suffix palindrome* of w.

Let $\mathit{Pals}(w)$ be the set of all center-distinct maximal palindromes where each element is encoded by a pair of its center and radius, namely,

$$\mathit{Pals}(w) = \left\{ (c, r) \;\middle|\; \begin{array}{l} w[c - r + 0.5 : c + r - 0.5] \text{ is a maximal palindrome} \\ \text{at center } c = 1, 1.5, 2, \ldots, n \end{array} \right\},$$

Also, let

$$\mathit{SPals}(w) = \{(c, r) \mid (c, r) \in \mathit{Pals}(w), c + r - 0.5 = |w|\},$$

namely, $\mathit{SPals}(w)$ represents the set of all suffix palindromes of w.

For example, let $w =$ `abbacabbba`. Then

$$\begin{aligned} \mathit{Pals}(w) = \{&(1, 0.5), (1.5, 0), (2, 0.5), (2.5, 2), (3, 0.5), (3.5, 0), (4, 0.5), (4.5, 0), \\ &(5, 3.5), (5.5, 0), (6, 0.5), (6.5, 0), (7, 0.5), (7.5, 1), (8, 2.5), (8.5, 1), \\ &(9, 0.5), (9.5, 0), (10, 0.5)\} \text{ and} \\ \mathit{SPals}(w) = \{&(8, 2.5), (10, 0.5)\}. \end{aligned}$$

Theorem 1 ([13]). *For any string w of length m, $\mathit{Pals}(w)$ can be computed in $O(m)$ time.*

Throughout this paper, we assume that the elements of $\mathit{Pals}(w)$ are sorted in increasing order of centers c. Actually, the algorithm of [13] computes the elements of $\mathit{Pals}(w)$ in this order.

In this paper, we tackle the following problem.

Problem 1 (Palindrome pattern matching, pal-matching in short). Given a text string t of length n and a pattern string p of length m, compute all positions i of t such that $\mathit{Pals}(p) = \mathit{Pals}(t[i : i + m - 1])$.

3 Linear-time Palindrome Pattern Matching Algorithm

To achieve a linear time solution to Problem 1, we design a pal-matching version of the Morris-Pratt algorithm [15].

Definition 1. *A* palindrome border *(*pal-border *in short) of a string p of length m is any integer j s.t. $0 \leq j < m$ and $\mathit{Pals}(p[1 : j]) = \mathit{Pals}(p[m - j + 1 : m])$.*

For example, the set of pal-borders of string $p =$ `aabcdaacdbcc`, is $\{7, 2, 1, 0\}$, since $\mathit{Pals}(\texttt{aabcdaa}) = \mathit{Pals}(\texttt{aacdbcc})$, $\mathit{Pals}(\texttt{aa}) = \mathit{Pals}(\texttt{cc})$, $\mathit{Pals}(\texttt{a}) = \mathit{Pals}(\texttt{c})$, and $\mathit{Pals}(\varepsilon) = \mathit{Pals}(\varepsilon)$.

Let $\mathcal{N}$ be the set of non-negative integers. For any string p of length m, let $\mathit{Pal\_Border}_p : \mathcal{N} \to \mathcal{N}$ be the function such that $\mathit{Pal\_Border}_p(m)$ equals the largest pal-border of string p. When clear from the context, we abbreviate $\mathit{Pal\_Border}_p$ as $\mathit{Pal\_Border}$. Since $\mathit{Pal\_Border}(m)$ is strictly smaller than m, we finally obtain 0 by iteratively applying the function *Pal_Border* to m. For any function $f : \mathcal{N} \to \mathcal{N}$ and any $m, k \in \mathcal{N}$, we define $f^k(m)$ as follows: $f^k(m) = f(m)$ if $k = 1$, and $f^k(m) = f(f^{k-1}(m))$ if $k \geq 2$. Similar to a standard border of a string [15], the following lemma holds.

Lemma 1. *For any string p of length m, let k be the smallest integer such that $\mathit{Pal\_Border}^k(m) = 0$. Then*

$$\mathit{Pal\_Border}(m), \mathit{Pal\_Border}^2(m), \ldots, \mathit{Pal\_Border}^k(m)$$

are all the pal-borders of p with $m > \mathit{Pal\_Border}(m) > \mathit{Pal\_Border}^2(m) > \cdots > \mathit{Pal\_Border}^k(m) = 0$.

Definition 2. *The* palindrome border array *(*pal-border array*) β_p of a string p of length m is an integer array of length m such that $\beta_p[i] = \mathit{Pal\_Border}_{p[1:i]}(i)$ for each $1 \leq i \leq m$.*

For example, for string $p =$ `aabbaa`, we have $\beta_p = [0, 1, 1, 2, 3, 4]$. When it is clear from the context, we abbreviate β_p as β.

In what follows, we present how to compute the pal-border array β_p of a given string p in linear time.

For any string w of length $m \geq 1$, let Lpal_w be an integer array of length m such that

$$\mathit{Lpal}_w[i] = \max\{i - k + 1 \mid w[k : i] = w[k : i]^R, 1 \leq k \leq i\}.$$

That is, the value of $\mathit{Lpal}_w[i]$ is equal to the length of the longest palindrome that ends at position i in w, for every $1 \leq i \leq m$[1]. Note that the above palindrome $w[k : i]$ is not necessarily a maximal palindrome at center $\frac{k+i}{2}$ in w.

For example, for string $w =$ `abbacabbba`, $\mathit{Lpal}_w = 1\ 1\ 2\ 4\ 1\ 3\ 5\ 7\ 3\ 5$.

The following lemma is a key to solve Problem 1 of pal-matching.

Lemma 2. *For any strings $w, z \in \Sigma^+$, $\mathit{Pals}(w) = \mathit{Pals}(z)$ iff $\mathit{Lpal}_w = \mathit{Lpal}_z$.*

Proof. ($\Longrightarrow$) We prove the claim by contradiction. Assume for contrary that $\mathit{Lpal}_w \neq \mathit{Lpal}_z$. Then there exists position i such that $\mathit{Lpal}_w[i] \neq \mathit{Lpal}_z[i]$. Assume w.l.o.g. that $\mathit{Lpal}_w[i] < \mathit{Lpal}_z[i]$. Let $k = (\mathit{Lpal}_z[i])/2$. The radius of the maximal palindrome centered at position $i - k + 0.5$ of w is less than k, however, that of the maximal palindrome centered at position $i - k + 0.5$ of z is at least k. This contradicts the assumption that $\mathit{Pals}(w) = \mathit{Pals}(z)$. Hence if $\mathit{Pals}(w) = \mathit{Pals}(z)$, then $\mathit{Lpal}_w = \mathit{Lpal}_z$.

($\Longleftarrow$) We prove the claim by contradiction and infinite descent. Assume for contrary that $\mathit{Pals}(w) \neq \mathit{Pals}(z)$. Then there exists center c such that $(c, r) \in \mathit{Pals}(w)$, $(c, u) \in \mathit{Pals}(z)$, and $r \neq u$. Assume w.l.o.g. that $r < u$.

In what follows, we consider position $j = \lceil c + u \rceil - 1$.

1. When $\mathit{Lpal}_w[j] < 2u$. Since $(c, u) \in \mathit{Pals}(z)$, $\mathit{Lpal}_z[j] \geq 2u$. This contradicts the assumption that $\mathit{Lpal}_w = \mathit{Lpal}_z$.

[1] The notion of $\mathit{Lpal}_w[i]$ was previously introduced in [8], denoted LPS[i] therein.

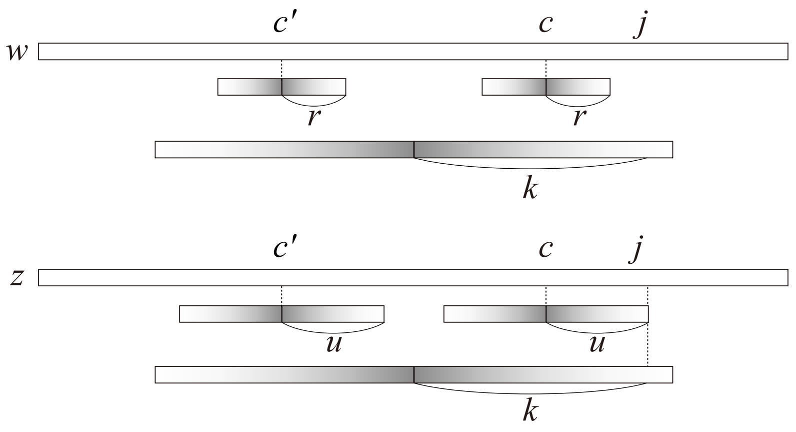

Fig. 1. Illustration for infinite descent in the proof of Lemma 2.

2. When $Lpal_w[j] \geq 2u$. Let $k = (Lpal_w[j])/2$. Then clearly w has a palindrome that is centered at $j-k+0.5$ and is of radius k. Also z has a palindrome that is centered at $j-k+0.5$ and is of radius k, since otherwise it contradicts the assumption that $Lpal_w = Lpal_z$. Then there exists center $c' < c$ such that $(c', r) \in Pals(w)$, $(c', u) \in Pals(z)$, and $r < u$. (See also Fig. 1.)
   The same must hold for those smaller centers, ad infinitum. However, this is impossible since w and z are finite strings.

Hence if $Lpal_w = Lpal_z$, then $Pals(w) = Pals(z)$. □

It is shown in [8] that $Lpal_w$ can be computed in linear time from $Pals(w)$. The following lemma is essentially the same as what is claimed in [8], but is more specifically tailored for our needs.

Lemma 3. *Let w be any string of length m. Given $Pals(w)$, $Lpal_w$ can be computed in $O(m)$ time, in an on-line fasion, from $Lpal_w[1]$ to $Lpal_w[m]$.*

Proof. For any position i of w with $1 \leq i \leq m$, the value of $Lpal_w[i]$ is equal to $2(i-c)+1$ where c is the smallest center of a maximal palindrome $(c, r) \in Pals(w)$ such that $c + r \geq i$. Hence we process the given string w from left to right.

Assume that we have computed $Lpal_w[1 : i]$ and let $(c, r) \in Pals(w)$ with $Lpal_w[i] = 2(i-c)+1$. Now we compute $Lpal_w[i+1]$. If $c + r \geq i+1$, then $Lpal_{w[1:i+1]} = 2((i+1)-c)+1$. Otherwise, we increment the value of c by 0.5 until satisfying $c + r \geq i+1$, where r is the radius of the maximal palindrome with the updated center c.

A pseudo-code of the algorithm is shown in Algorithm 1. The correctness should be clear from the above arguments. Note that the value of c does not decrease and does not exceed the value of i. Also, (c, r) can be picked up from $Pals(w)$ in constant time at each step, since $Pals(w)$ is sorted in increasing order of c. Consequently the time complexity is linear in m. □

Algorithm 1: On-line algorithm to compute $Lpal_w$ of w.

```
Input: String $w$ of length $m$.
Output: $Lpal_w[1:m]$.
1 compute $Pals(w)$;
2 $c \leftarrow 1$; let $(c, r) \in Pals(w)$;
3 for $i \leftarrow 1$ to $m$ do
4     while $c + r < i$ do
5         $c \leftarrow c + 0.5$; let $(c, r) \in Pals(w)$;
6     $Lpal_w[i] \leftarrow 2(i - c) + 1$;
7 return $Lpal_w[1:m]$;
```

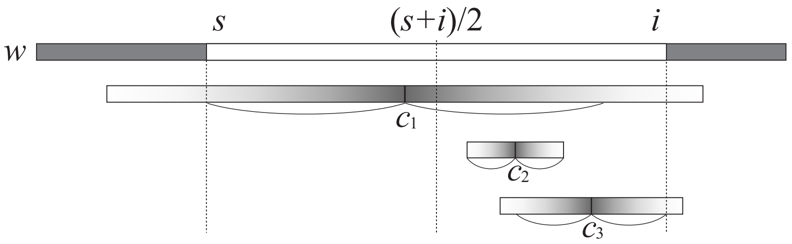

Fig. 2. If (c_3, r_3) is the maximal palindrome in $Pals(w)$ such that c_3 is the smallest center satisfying $c_3 \geq (s+i)/2$ and $c_3 + r_3 \geq i$, c_3 is the active center for s and i, and $Lpal_{w[s:i]}[i-s+1] = 2(i-c_3)+1$. Note that c_1 is not the active center for s and i since $c_1 < (s+i)/2$.

Let w be any string of length m, and let s and i be any integers with $1 \leq s \leq i \leq m$. Here we consider computing $Lpal_{w[s:i]}[i-s+1]$ from $Pals(w)$. By the definition of $Lpal$, the value of $Lpal_{w[s:i]}[i-s+1]$ is equal to $2(i-c)+1$, where (c, r) is the maximal palindrome in $Pals(w)$ such that c is the smallest center satisfying $c \geq (s+i)/2$ and $c + r \geq i$ (See also Fig. 2). We call this center c *the active center* for s and i w.r.t. w, and denote it by $AC_w(s, i)$. It holds that $Lpal_{w[s:i]}[i-s+1] = 2(i - AC_w(s, i)) + 1$.

Lemma 4. *Let w be any string of length m. For any integers s, i, s', i' with $1 \leq s \leq i \leq m$ and $1 \leq s' \leq i' \leq m$, if $s \leq s'$ and $i \leq i'$, then $AC_w(s, i) \leq AC_w(s', i')$.*

Proof. Assume for contrary that $AC_w(s, i) > AC_w(s', i')$. Since $AC_w(s, i) \leq i$, $AC_w(s', i') < i$. Let $(AC_w(s', i'), r) \in Pals(w)$. It follows from $AC_w(s', i') \geq (s'+i')/2 \geq (s+i)/2$ and $AC_w(s', i') + r \geq i' \geq i$ that $AC_w(s', i') \geq (s+i)/2$ and $AC_w(s', i') + r \geq i$. However this contradicts that $AC_w(s, i)$ is the active center for s and i w.r.t. w. □

In the algorithms which follow, we will need to know the value of $Lpal_{w[s:i]}[i-s+1]$ for some s and i. It seems difficult to compute $Lpal_{w[s:i]}[i-s+1]$ in constant time for "randomly" chosen s and i, with $O(m)$-time preprocessing.

Algorithm 2: Algorithm to compute β_p of a given string p.

```
Input: String $p$ of length $m$.
Output: Pal-border array $\beta_p[1:m]$.
1  compute $Pals(p)$ and $Lpal_p[1:m]$;
2  $\beta_p[1] \leftarrow 0$;
3  $j \leftarrow 0$; $c \leftarrow 0$;
4  for $i \leftarrow 2$ to $m$ do
5      while true do
6          $c \leftarrow \max\{c, i - j/2\}$; let $(c, r) \in Pals(p)$;
7          while $c + r < i$ do /* Shift $c$ to $AC_p(i - j, i)$.  */
8              $c \leftarrow c + 0.5$; let $(c, r) \in Pals(p)$;
           /* $2(i - c) + 1 = Lpal_{p[i-j:i]}[j + 1]$.  */
9          if $Lpal_p[j + 1] = 2(i - c) + 1$ then break;
10         $j \leftarrow \beta_p[j]$;
11     $j \leftarrow j + 1$;
12     $\beta_p[i] \leftarrow j$;
13 return $\beta_p[1:m]$;
```

Nevertheless, Lemma 4 suggests that, if s and i monotonically increase from 1 to m, then the total cost for computing $Lpal_{w[s:i]}[i - s + 1]$ for all s and i never exceeds the number of the centers in w, which is $2m - 1$. The point is that all the following algorithms only require to compute $Lpal_{w[s:i]}[i - s + 1]$ for monotonically increasing positions s and i, with $1 \leq s \leq i \leq m$.

Lemma 5. *For any string p of length m, β_p can be computed in $O(m)$ time.*

Proof. Algorithm 2 describes our algorithm. This algorithm is mostly the same as the linear-time algorithm for computing a standard border array of a string [15], except that we match the values of $Lpal$ instead of characters.

We firstly compute $Pals(p)$ and $Lpal_p[1 : m]$. This takes $O(m)$ time by Theorem 1 and Lemma 3. Then we compute $\beta_p[1 : m]$ in ascending order. Consider the i-th iteration of the **for** loop of Line 4. Here we have computed $\beta_p[1 : i-1]$, and variable j is set to be $\beta_p[i-1]$. Next we compute $Lpal_{p[i-j:i]}[j+1]$ by shifting the current center c right to $AC_p(i - j, i)$. If $Lpal_p[j + 1] = Lpal_{p[i-j:i]}[j + 1]$, $\beta_p[i] = j + 1$. Otherwise, we set j to be $\beta_p[j]$ and check again if $Lpal_p[j + 1] = Lpal_{p[i-j:i]}[j + 1]$ or not. The above procedure is repeated until j, such that $Lpal_p[j + 1] = Lpal_{p[i-j:i]}[j + 1]$, is found. Note that we break this loop at the latest when $j = 0$, since $Lpal_p[1] = Lpal_{p[i:i]}[1] = 1$.

In each iteration of the **for** loop of Line 4, the value of j increases by at most 1. Since each execution of the **while** loop of Line 5 decreases the value of j at least 1 and $j \geq 0$, the **while** loop of Line 5 is executed at most m times in total. Moreover, since the value of c does not decrease and does not exceed the value of i, the total cost of the **while** loop of Line 7 is $O(m)$. Therefore Algorithm 2 runs in time linear in m. □

Algorithm 3: Algorithm to solve pal-matching problem in linear time.

```
Input: Text string $t$ of length $n$ and pattern string $p$ of length $m$.
Output: All positions $i$ of $t$ such that $t[i : i + m - 1]$ pal-matches $p$.
1  compute $Pals(t)$, $Lpal_p[1 : m]$, and $\beta_p[1 : m]$;
2  $j \leftarrow 0$; $c \leftarrow 0$;
3  for $i \leftarrow 1$ to $n$ do
4      while true do
5          $c \leftarrow \max\{c, i - j/2\}$; let $(c, r) \in Pals(t)$;
6          while $c + r < i$ do /* Shift $c$ to $AC_t(i - j, i)$.          */
7              $c \leftarrow c + 0.5$; let $(c, r) \in Pals(t)$;
           /* $2(i - c) + 1 = Lpal_{t[i-j:i]}[j + 1]$.                    */
8          if $Lpal_p[j + 1] = 2(i - c) + 1$ then break;
9          $j \leftarrow \beta_p[j]$;
10     $j \leftarrow j + 1$;
11     if $j = m$ then
12         $j \leftarrow \beta_p[j]$; report $i - m + 1$;
```

Theorem 2. *The pal-matching problem (Problem 1) can be solved in $O(n+m)$ time.*

Proof. Algorithm 3 describes our algorithm. This algorithm is a pal-matching version of the Morris-Pratt algorithm [15].

We firstly compute $Pals(p)$ by Algorithm 1 and $Lpal_p[1 : m]$ by Algorithm 2 in $O(m)$ time, and $Pals(t)$ in $O(n)$ time. Consider the i-th iteration of the **for** loop of Line 3. Here variable j represents an integer such that $p[1 : j]$ and $t[i - j : i - 1]$ pal-match. Next we compute $Lpal_{t[i-j:i]}[j + 1]$ by shifting the current center c right to $AC_t(i - j, i)$. If $Lpal_p[j + 1] = Lpal_{t[i-j:i]}[j + 1]$, we break the **while** loop of Line 4. Otherwise, we set j to be $\beta_p[j]$ and check again if $Lpal_p[j + 1] = Lpal_{t[i-j:i]}[j + 1]$ or not. The above procedure is repeated until j, such that $Lpal_p[j + 1] = Lpal_{t[i-j:i]}[j + 1]$, is found. Note that we break this loop at the latest when $j = 0$, since $Lpal_p[1] = Lpal_{t[i:i]}[1] = 1$. After breaking the **while** loop of Line 4, we increment j by 1, and if j becomes m, the algorithm reports that $t[i - m + 1 : i]$ and $p[1 : m]$ pal-match.

In each iteration of the **for** loop of Line 3, the value of j increases by at most 1. Since each execution of the **while** loop of Line 4 decreases the value of j at least 1 and $j \geq 0$, the **while** loop of Line 4 is executed at most n times in total. Moreover, since the value of c does not decrease and does not exceed the value of i, the total cost of the **while** loop of Line 6 is $O(n)$. Therefore Algorithm 3 runs in $O(n + m)$ time. □

4 Palindrome Suffix Trees

In this section, we consider an indexing structure for pal-matching. We propose a new data structure called *palindrome suffix trees* (*pal-suffix trees* in short).

$t =$ a b b a b b c b c	
$Lpal_{t[1:9]}[1:9] =$	1 1 2 4 3 5 1 3 3
$Lpal_{t[2:9]}[1:8] =$	1 2 1 3 5 1 3 3
$Lpal_{t[3:9]}[1:7] =$	1 1 3 2 1 3 3
$Lpal_{t[4:9]}[1:6] =$	1 1 2 1 3 3
$Lpal_{t[5:9]}[1:5] =$	1 2 1 3 3
$Lpal_{t[6:9]}[1:4] =$	1 1 3 3
$Lpal_{t[7:9]}[1:3] =$	1 1 3
$Lpal_{t[8:9]}[1:2] =$	1 1
$Lpal_{t[9:9]}[1:1] =$	1

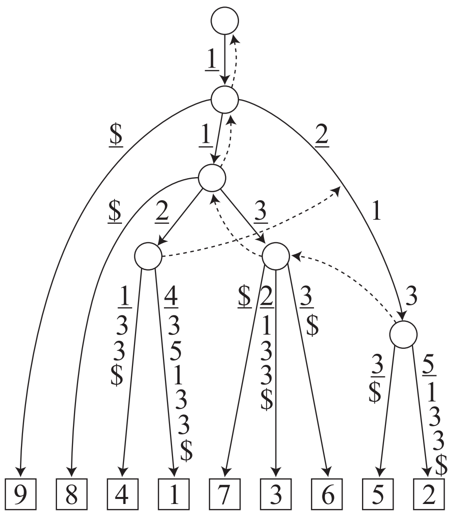

Fig. 3. Illustration of $Pal\_ST(t)$ for string $t =$ `abbabbcbc`. The solid arrows represent the edges, and the broken arrows do the suffix links. The path from the root to each leaf s spells out $Lpal_{t[s:9]}[1:s]\$$.

The pal-suffix tree of a string t, denoted $Pal\_ST(t)$, is a compacted trie which represents $Lpal_{t[s:n]}[1:n-s+1]$ for all the suffixes $t[s:n]$ of t, where n is the length of t and $1 \leq s \leq n$. Each internal node of $Pal\_ST(t)$ has at least two children, and the labels of two distinct out-going edges of each internal node must start with distinct non-negative integers. Moreover, for $Pal\_ST(t)$ to have exactly n leaves, we use the following convention: Each leaf of $Pal\_ST(t)$ is uniquely labeled with integer s ($1 \leq s \leq n$) in such a way that the path from the root to leaf s spells out $Lpal_{t[s:n]}[1:n-s+1]\$$, where \$ is a special end-marker. The *length* of a node v, denoted $len(v)$, is the length of $Lpal$ represented by v. Fig. 3 illustrates Pal_ST(`abbabbcbc`).

Notice that there are $O(n)$ distinct values for the elements of $Lpal_t[1:n]$. For instance, consider $t = (\mathtt{ab})^{\frac{n}{2}}$. Then $Lpal_t[1:n] = 1\ 1\ 3\ 3 \cdots n{-}1\ n{-}1$. This suggests that an internal node of $Pal\_ST(t)$ might have $O(n)$ children. However, the following lemma holds.

Lemma 6. *For any string t, each node of $Pal\_ST(t)$ has at most σ children, where σ is the alphabet size.*

Proof. For any string w, let $S(w) = SPals(w) \cup \{(|w|+0.5, 0)\} - \{(|w|/2 + 0.5, |w|/2)\}$. To show the lemma, we consider the following claim.

Claim. Let w and z be any strings of length i s.t. $Pals(w) = Pals(z)$. For any integers j, k with $1 \leq j \leq i$, $1 \leq k \leq i$ and $(\frac{i+j+1}{2}, \frac{i-j}{2}), (\frac{i+k+1}{2}, \frac{i-k}{2}) \in S(w)$, if $w[j] = w[k]$ then $z[j] = z[k]$.

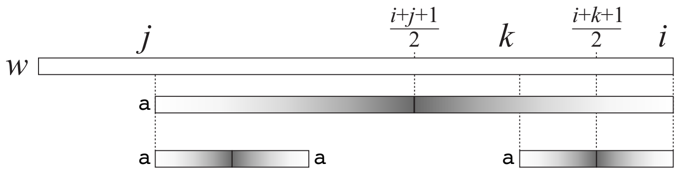

Fig. 4. Illustration for the proof of Claim in Lemma 6.

Proof of Claim. When $j = k$, it is clear the claim holds. Then we consider the case $j \neq k$. Assume w.l.o.g. that $j < k$. Since $w[j+1 : i+j-k] = w^R[k+1 : i] = w[k+1 : i] = w^R[j+1 : i+j-k]$, $w[j+1 : i+j-k]$ is a palindrome. It follows from $w[j] = w[k]$ and $w[j+1 : i] = w^R[j+1 : i]$ that $w[j] = w[k] = w[i+j+1-k]$. Putting $w[j+1 : i+j-k] = w^R[j+1 : i+j-k]$ and $w[j] = w[i+j+1-k]$ together, we get $w[j : i+j+1-k]$ is a palindrome (See also Fig. 4). Since $Pals(w) = Pals(z)$, $z[j : i+j+1-k]$ and $z[j+1 : i]$ are palindromes, and thus $z[j] = z[i+j+1-k] = z[k]$. Hence the claim holds.

Consider any substring w of length i of t. We introduce an equivalence relation on $S(w)$ such that

$$(\frac{i+j+1}{2}, \frac{i-j}{2}) \equiv (\frac{i+k+1}{2}, \frac{i-k}{2}) \iff w[j] = w[k],$$

where $1 \leq j \leq i$, $1 \leq k \leq i$, and $(\frac{i+j+1}{2}, \frac{i-j}{2}), (\frac{i+k+1}{2}, \frac{i-k}{2}) \in S(w)$. By definition, there are at most σ equivalence classes w.r.t. $\equiv$. Consider any substring z of t with $Pals(z) = Pals(w)$. Due to the above claim, the equivalence classes on $S(z)$ are identical to those on $S(w)$.

Let v be any node of $Pal\_ST(t)$, and assume that the path from the root to v spells out $Lpal_w$. Note that every substring z of t that pal-matches w is represented by the same node v in $Pal\_ST(t)$, since it has the same $Lpal$ values as w, i.e., $Lpal_w = Lpal_z$. Therefore, the number of children of v is at most $d+1$, where d is the number of equivalence classes on $S(w)$, which is bounded by σ. Hence the lemma holds. □

In order to implement $Pal\_ST(t)$ with $O(n)$ space, we encode the label of each edge as follows. Assume that there is an edge of $Pal\_ST(t)$ labeled with x, where x is a sequence of positive integers. We encode x by a triple $(x[1], q, |x|)$, where $x[1]$ is the first element of x, q is a position of text t such that $x = Lpal_{t[s:n]}[q-s+1 : q-s+|x|]$ for some $1 \leq s \leq n$, and $|x|$ is the length of the edge label. See Fig. 3 and focus on the edge which is labeled with 2 1 3. Choosing $s = 2$, the label is encoded by $(2, 3, 3)$ as $q = 3$, $|x| = 3$, and $Lpal_{t[2:9]}[2 : 4] = 2\ 1\ 3$. In Fig. 3, the first element of each edge label is shown underlined.

Theorem 3. *Provided that $Pal\_ST(t)$ and $Pals(t)$ are already computed, the pal-matching problem (Problem 1) can be solved in $O(m \log \sigma + r)$ time, where r is the output size.*

Proof. We compute $Lpal_p$ using Algorithm 1 in $O(m)$ time. Then we search $Pal\_ST(t)$ for $Lpal_p[1:m]$. Assume that $Lpal_p[1:j]$ matches the label of an out-going edge of the root node of $Pal\_ST(t)$, with some $1 \leq j < m$. Assume the edge label is encoded as $(Lpal_{t[q:n]}[1], q, j)$, where $Lpal_{t[q:n]}[1:j] = Lpal_p[1:j]$. Let v be the node that represents $Lpal_{t[q:n]}[1:j]$. Assume that there is an out-going edge of v, which is labeled with $(Lpal_{t[q'-j:n]}[j+1], q', j')$, where $Lpal_{t[q'-j:n]}[j+1] = Lpal_p[j+1]$ and $j' \geq 2$. This edge can be found in $O(\log \sigma)$ time by Lemma 6. Now we have to check whether $Lpal_{t[q'-j:n]}[j+2] = Lpal_p[j+2]$. Although q' is *not* necessarily equal to $q+j$, we can compute $Lpal_{t[q'-j:n]}[j+2]$ as follows: By the definition of $Pal\_ST(t)$ it holds that $Lpal_{t[q'-j:n]}[1:j+1] = Lpal_{t[q:n]}[1:j+1]$, which implies that $AC_t(q'-j, q') = AC_t(q, q+j) + q' - (q+j)$. As described in Section 3, we can compute $Lpal_{t[q'-j:n]}[j+2]$ by shifting the current center from $AC_t(q'-j, q')$ to $AC_t(q'-j, q'+1)$. Moreover, $Lpal_{t[q'-j:n]}[j+2] = Lpal_p[j+2]$ iff $AC_t(q'-j, q'+1) - AC_t(q'-j, q') = AC_p(1, j+2) - AC_p(1, j+1)$. In light of this, the total cost for computing such values of $Lpal$ is bounded by the cost for computing $Lpal_p$, which is $O(m)$. We continue the above procedure until either we find $Lpal_p$ in $Pal\_ST(t)$ or we find a mismatch. This takes $O(m \log \sigma)$ time. If $Lpal_p$ is found, then we traverse the sub-tree rooted at the (possibly implicit) node that represents $Lpal_p$, and report the id of the leaves in the sub-tree, in $O(r)$ time. □

4.1 Constructing Palindrome Suffix Trees

We employ Ukkonen's on-line construction techniques for suffix trees [17]. Here let us briefly review the behavior of the Ukkonen's algorithm. The algorithm processes the characters of a given string t of length n in ascending order. After processing the $(i-1)$-th character of t, the algorithm has constructed the suffix tree of $t[1:i-1]$. Now the algorithm waits for the next i-th character on the location which represents the longest suffix $t[s:i-1]$ of $t[1:i-1]$ that matches a substring of $t[1:i-2]$, with some $2 \leq s \leq i$. Let us call this location on the path *the active point* for $i-1$. Next the algorithm obtains the i-th character $t[i]$. If we can transit from the active point for $i-1$ with $t[i]$, then the active point for i is the location that represents $t[s:i]$. Otherwise, the algorithm creates a new edge from the active point for $i-1$ leading to a new leaf node, with edge label $t[i:n]$. After that, the algorithm finds the location which represents $t[s+1:i-1]$ by using a suffix link, in amortized constant time. The above procedure is repeated until the active point for i is found. Readers are referred to [17] for more details of the Ukkonen algorithm.

In the sequel, we show main technical issues of our algorithm to construct $Pal\_ST(t)$.

Suffix Links. Let v be any node of $Pal\_ST(t)$, and assume that the path from the root to v spells out $Lpal_w$ for some substring w of t. *The suffix link* of node v is an auxiliary edge from node v to node u, such that the path from the root to u spells out $Lpal_{w[2:|w|]}$. For example, see Fig. 3, and focus on the node which

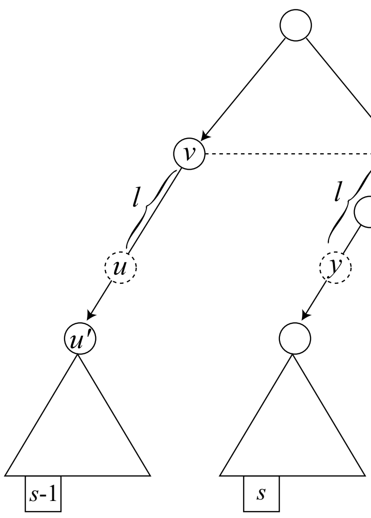

Fig. 5. Illustration of maintenance of the active point. u is the active point for $i-1$, and y is a candidate for the active point for i.

represents 1 2 1 3. The suffix link of this node points to the node which represents 1 1 3. This is because there exists a substring `bbab` with $Lpal_{\texttt{bbab}} = 1\ 2\ 1\ 3$, and $Lpal_{\texttt{bab}} = 1\ 1\ 3$.

Unlike the case of suffix trees, the node u, which is to be pointed by the suffix link of some node v, is *not* always explicit in $Pal\_ST(t)$. For example, see Fig 3. The suffix link of the node which represents 1 1 2 is illustrated to point to the implicit node which represents 1 2. In such a case, we set the suffix link of node v to the child node u' of implicit node u, and record the length of the partial edge label from u to u'. This way we can access from node v to the location for u in constant time. In the above example, the suffix link of node 1 1 2 is implemented to point to node 1 2 1 3, with auxiliary value 2 which is the length of the partial label from implicit node 1 2 to node 1 2 1 3. The same technique was used in [3] to implement the suffix links of *parameterized suffix trees*.

Maintaining Active Point. Assume that we have constructed $Pal\_ST(t[1:i-1])$ for given string t, for some $1 \leq i \leq n$. Assume that the active point for $i-1$ is on an implicit node u. Let v be the explicit parent node of u, and let u' be the explicit child node of v, i.e., u is on the edge from v to u'. Let x be the label of the edge from v to u', and let ℓ be the length of the partial edge label from v to u. Then, the active point for $i-1$, the implicit node u, is represented by $(v, x[1], s-1+len(v), \ell)$, where $x[1]$ is the first element of x and $s-1$ is a position of t such that $Lpal_{t[s-1:n]}[len(v)+1 : len(v)+\ell] = x[1:\ell]$.

Similarly to construction of suffix trees, we look for the active point for i from the active point for $i-1$, i.e., the implicit node u. See Fig. 5. In so doing, we use the suffix link of node v. Consider any leaf $s-1$ in the subtree rooted at v. Let q be the node we have reached by the suffix link of node v. Now we want to look for a (possibly implicit) child y of q such that the subtree rooted at y has leaf s and $len(y) = len(u) - 1 = len(q) + \ell$. The difficulty we face is that $x[1 : \ell] = Lpal_{t[s-1:n]}[len(v) + 1 : len(v) + \ell]$ may not be equal to $Lpal_{t[s:n]}[len(q) + 1 : len(q) + \ell]$. This happens when there exists an integer k, $len(v)+1 \leq k \leq len(v)+\ell$, such that $Lpal_{t[s-1:n]}[k] = k$. For example, see Fig 3. The edge leading to leaf 2 is labeled with 5 1 3 3 \$, while the edge leading to leaf 3 is labeled with 2 1 3 3 \$. This is because $Lpal_{t[2:9]}[5] = 5$.

Nevertheless, we can efficiently locate y starting from q, as follows. Since $x[1] = Lpal_{t[s-1:n]}[len(v) + 1]$, we can calculate $AC_t(s - 1, s - 1 + len(v))$ in constant time. Since $len(q) = len(v) - 1$, we can compute $Lpal_{t[s:n]}[len(q) + 1 : len(q)+\ell]$ in $O(AC_t(s, s+len(q)) - AC_t(s-1, s+len(q))+\ell)$ time, as described in Section 3. Then we can find y in $O(\ell \log \sigma)$ time, since there can be at most $\ell - 1$ explicit nodes in the path from q to y. We check whether y is the active point for i or not, and if not, we repeat the above procedure until the active point for i is found. The total cost of the above operations, after constructing $Pal\_ST(t)$, is $O(n \log \sigma)$.

Consequently, we obtain the following result.

Theorem 4. *For any string t of length n, $Pal\_ST(t)$ can be constructed in $O(n \log \sigma)$ time, where σ is the alphabet size.*

5 Conclusions and Future Work

Palindromes in strings have widely been studied both in theoretical and practical contexts, such as in word combinatorics and in bioinformatics. In this paper, we presented linear-time algorithms to solve a new problem called the palindrome pattern matching problem. The first algorithm is a Morris-Pratt type algorithm, and the second one is a suffix-tree type algorithm.

In practical applications such as DNA and RNA sequence analysis, it is desired to cope with *gapped palindromes* which have a spacer between the left and right arms of the palindromes. Several versions of gapped palindromes have been introduced and studied [9, 12, 10]. Our future work includes development of efficient solutions to a gapped-palindromes version of the palindrome pattern matching problem.

References

1. Allouche, J.P., Baake, M., Cassaigne, J., Damanik, D.: Palindrome complexity. Theoretical Computer Science 292(1), 9–31 (2003)
2. Anisiu, M.C., Anisiu, V., Kása, Z.: Total palindrome complexity of finite words. Discrete Mathematics 310(1), 109–114 (2010)
3. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 52(1), 28–42 (1996)

4. Brlek, S., Hamel, S., Nivat, M., Reutenauer, C.: On the palindromic complexity of infinite words. International Journal of Foundations of Computer Science 15(2), 293–306 (2004)
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., DIGITAL System Research Center (1994)
6. Droubay, X., Justin, J., Pirillo, G.: Episturmian words and some constructions of de Luca and Rauzy. Theoretical Computer Science 255(1–2), 539–553 (2001)
7. Glen, A., Justin, J., Widmer, S., Zamboni, L.Q.: Palindromic richness. European Journal of Combinatorics 30(2), 510–531 (2009)
8. Groult, R., Prieur, É., Richomme, G.: Counting distinct palindromes in a word in linear time. Information Processing Letters 110(20), 908–912 (2010)
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
10. Hsu, P.H., Chen, K.Y., Chao, K.M.: Finding all approximate gapped palindromes. In: Proc. ISAAC 2009. LNCS, vol. 5878, pp. 1084–1093 (2009)
11. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting and verifying maximal palindromes. In: Proc. SPIRE 2010. LNCS, vol. 6393, pp. 135–146 (2010)
12. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theoretical Computer Science 410(51), 5365–5373 (2009)
13. Manacher, G.: A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. Journal of the ACM 22(3), 346–351 (1975)
14. Massé, A.B., Brlek, S., Frosini, A., Labbé, S., Rinaldi, S.: Reconstructing words from a fixed palindromic length sequence. In: Proc. TCS 2008. IFIP, vol. 273, pp. 101–114 (2008)
15. Morris, J.H., Pratt, V.R.: A linear pattern-matching algorithm. Tech. Rep. 40, University of California, Berkeley (1970)
16. Restivo, A., Rosone, G.: Burrows-Wheeler transform and palindromic richness. Theoretical Computer Science 410(30–32), 3018–3026 (2009)
17. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)