PalFM-Index: FM-Index for Palindrome Pattern Matching

Shinya Nagashita
Kyushu Institute of Technology, Fukuoka, Japan

Tomohiro I
Kyushu Institute of Technology, Fukuoka, Japan

Abstract

The palindrome pattern matching (pal-matching) is a kind of generalized pattern matching, in which two strings $x$ and $y$ of same length are considered to match (pal-match) if they have the same palindromic structures, i.e., for any possible $1 \leq i < j \leq |x| = |y|$, $x[i..j]$ is a palindrome if and only if $y[i..j]$ is a palindrome. The pal-matching problem is the problem of searching for, in a text, the occurrences of the substrings that pal-match with a pattern. Given a text $T$ of length $n$ over an alphabet of size $\sigma$, an index for pal-matching is to support, given a pattern $P$ of length $m$, the counting queries that compute the number $\text{occ}$ of occurrences of $P$ and the locating queries that compute the occurrences of $P$. The authors in [I et al., Theor. Comput. Sci., 2013] proposed an $O(n \lg n)$-bit data structure to support the counting queries in $O(m \lg \sigma)$ time and the locating queries in $O(m \lg \sigma + \text{occ})$ time. In this paper, we propose an FM-index type index for the pal-matching problem, which we call the PalFM-index, that occupies $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space and supports the counting queries in $O(m)$ time. The PalFM-indexes can support the locating queries in $O(m + \Delta \text{occ})$ time by adding $n \Delta \lg n + n + o(n)$ bits of space, where $\Delta$ is a parameter chosen from $\{1, 2, \ldots, n\}$ in the preprocessing phase.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Palindrome matching, Generalized string pattern matching, Indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.23

Related Version Previous Version: https://arxiv.org/abs/2206.12600

Funding This work was supported by JSPS KAKENHI (Grant Number 19K20213). Tomohiro I: KAKENHI (Grant Numbers 19K20213).

1 Introduction

A palindrome is a string that can be read same backward as forward. Palindromic structures in a string are one of the most fundamental structures in the string and have been extensively studied. For example, it is known that any string $w$ contains at most $|w| + 1$ distinct palindromic substrings [6], and the strings reaching the maximum values have some intriguing properties [15, 28]. Another concept regarding palindromic structures is the palindrome complexity [1, 4, 2], which is the number of distinct palindromic substrings of a given length in a string.

Instead of thinking about distinct palindromic substrings, one might be interested in occurrences of palindromic substrings. The palindromic structures in such a sense are captured by the maximal palindromes from all possible “centers” in a string. Manacher’s algorithm [26], originally proposed for computing a prefix-palindrome, can be extended to compute all the maximal palindromes in $O(|w|)$ time for a string $w$. The authors in [18] considered the problem of inferring strings from a given set of maximal palindromes and showed that the problem can be solved in $O(|w|)$ time.
In [19], a new concept called palindrome pattern matching was introduced as a generalized pattern matching. Two strings $x$ and $y$ of the same length are said to palindromic pattern match (pal-match in short) iff they have the same palindromic structures, i.e., the following condition holds: for any possible $1 \leq i < j \leq |x| = |y|$, $x[i..j]$ is a palindrome iff $y[i..j]$ is a palindrome. We remark that $x$ and $y$ themselves are not necessarily palindromes. The palindrome pattern matching has potential applications to genomic analysis, in which some palindromic structures play an important role to estimate RNA secondary structures [21].

The pal-matching problem is to search for, in a text, the occurrences of the substrings that pal-match with a pattern. Given a text $T$ of length $n$ and a pattern $P$ of length $m$, a Morris-Pratt type algorithm for solving the pal-matching problem in $O(n)$ time was proposed in [19]. The method in [19] is based on the $lpal$-encoding of a string $w$, denoted as $lpal_w$, that is the integer array of length $|w|$ such that $lpal_w[i]$ is the length of the longest suffix palindrome of $w[1..i]$. The $lpal$-encoding is helpful because two strings $x$ and $y$ pal-match iff $lpal_x = lpal_y$. When $T$ is large and static, and patterns come online later, one might think of preprocessing $T$ to construct an index for pal-matching. An index for pal-matching is to support the counting queries that compute the number $occ$ of occurrences of $P$ and the locating queries that compute the occurrences of $P$. For this purpose, I et al. [19] proposed the palindrome suffix tree of $T$, which is a compacted tree of the $lpal$-encoded suffixes of $T$. The palindrome suffix tree takes $O(n \lg n)$ bits of space and supports the counting queries in $O(m \lg \sigma)$ time and the locating queries in $O(m \lg \sigma + occ)$ time, where $\sigma$ is the size of the alphabet from which characters in $T$ are taken and $occ$ is the number of occurrences.

In this paper, we present a new index, named the PalFM-index, by applying the technique of the FM-index [7] to the pal-matching problem. In so doing we introduce a new encoding, named the $ssp$-encoding, that is based on the non-trivial shortest suffix-palindrome of each prefix. In contrast to the $lpal$-encoding, the $ssp$-encoding has a good property to design the PalFM-index. The PalFM-index occupies $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space and supports the counting queries in $O(m + \Delta occ)$ time. The locating queries can be supported in $O(m + \Delta occ)$ time by adding $\frac{n}{\Delta} \lg n + n + o(n)$ bits of space, where $\Delta$ is a parameter chosen from $\{1, 2, \ldots, n\}$ in the preprocessing phase.

1.1 Related work

One of the well-studied algorithmic problems related to palindromes is factorizing a string into non-empty palindromes, or in other words, recognizing a string that is obtained by concatenating a certain number of non-empty palindromes [26, 24, 12, 9, 20, 25, 3, 29]. The combinatorial properties discovered during tackling this factorization problem are useful to work on palindromes-related problems.

Developing techniques of designing space-efficient indexes for generalized pattern matching is of great interest. Our PalFM-index was inspired by that of Kim and Cho [23], which is a simplified version of the FM-index for parameterized pattern matching [13]. Indexes based on the FM-index for other generalized pattern matching problems were considered in [14, 11, 22].

2 Preliminaries

2.1 Notations

An integer interval $\{i, i+1, \ldots, j\}$ is denoted by $[i..j]$, where $[i..j]$ represents the empty interval if $i > j$. 
Let Σ be a finite alphabet, a set of characters. 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, |ε| = 0. The concatenated string of two strings x and y are denoted as x · y or simply xy. The i-th character of a string w is denoted by w[i] for 1 ≤ i ≤ |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 ≤ i ≤ j ≤ |w|, i.e. w[i..j] = w[i]w[i+1]···w[j]. For convenience, let w[i..j] = ε if i > j. A substring of the form w[1..j] (resp. w[i..|w|]) is called a prefix (resp. suffix) of w and denoted as w[..j] (resp. w[..i]) in shorthand. Note that w is a substring/prefix/suffix of any string w. A substring of w is called proper if it is not w itself. When needed we use parentheses to indicate positions in a concatenated string, for example, (xy)[i] refers to the i-th character of the string xy. Hence, (xy)[i] should be distinguished from xy[i], which can be interpreted as the concatenated string of x and y[i].

Let < denote the total order over an alphabet we consider. In particular, we will consider strings over a set consisting of integers and ∞, in which natural total order based on their values is employed. We extend < to denote the lexicographic order of strings over the alphabet. For any strings x and y that do not match, we say that x is lexicographically smaller than y and denote it by x < y iff x[i+1] < y[i+1] for largest integer i with x[i..] = y[i..], where we assume that x[i+1] or y[i+1] refers to the lexicographically smallest character $\$ if it points to out of bounds.

For any string w, let $w^R$ denote the reversed string of w, that is, $w^R = w[|w|]···w[2]w[1]$. A string w is called a palindrome if $w = w^R$. 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 $j = |w|$.

Two strings x and y of same length are said to palindromic pattern match (pal-match in short) if they have the same palindromic structures, i.e., the following condition holds: for any possible 1 ≤ i < j ≤ |x| = |y|, x[i..j] is a palindrome iff y[i..j] is a palindrome. For example, abcbaaca and bcacbbdb pal-match since their palindromic structures coincide (see Figure 1). Note that pal-matching induces a substring consistent equivalent relation [27], i.e., if x and y pal-match then x[i..j] and y[i..j] pal-match for any possible 1 ≤ i < j ≤ |x| = |y|.

The pal-matching problem is to search for, in a text string T, the occurrences of the substrings that pal-match with a pattern P. In the pal-matching problem, an occurrence of P refers to a position $i$ such that $T[i..i+|P| - 1]$ and P pal-match. Throughout this paper we consider indexing a text T of length n over an alphabet Σ of size σ.

Figure 1 Illustration of the palindromic structures for pal-matching strings abcbaaca and bcacbbdb. Check that the radii of their maximal palindromes for all possible centers, which are illustrated by two-headed arrows, coincide.
2.2 Toolbox

As a component of our PalFM-index, we use a data structure for a string \( w \) over an integer alphabet \( U \) supporting the following queries.

- \( \text{rank}_w(i, c) \): return the number of occurrences of character \( c \in U \) in \( w[i..j] \).
- \( \text{select}_w(i, c) \): return the \( i \)-th smallest position of the occurrences of character \( c \in U \) in \( w \).
- \( \text{rangeCount}_w(i, j, c, d) \): return the number of the occurrences of any character in \([e..d] \subseteq U \) in \( w[i..j] \).

The Wavelet tree \([17]\) supports these queries in \( O(\lg |U|) \) time using \( |w|\mathcal{H}_0(w) + o(|w| \lg |U|) \) bits of space, where \( \mathcal{H}_0(w) = O(\lg |U|) \) is the 0-th order empirical entropy of \( w \). The subsequent studies \([8, 16]\) improved the complexities, resulting in the following theorem.

\[ \text{Theorem 1 ([16])}. \] For a string \( w \) over an integer alphabet \( U \), there is a data structure in \( |w|\mathcal{H}_0(w) + o(|w|) \) bits of space that supports \( \text{rank}, \text{select} \) and \( \text{rangeCount} \) in \( O(1 + \frac{\lg |U|}{\lg \lg |U|}) \) time.

We also use a data structure for the Range Maximum Queries (RMQs) over an integer array \( V \). Given an interval \([i..j]\) over \( V \), a query \( \text{RMQ}_V(i, j) \) returns a position in \([i..j]\) that has the maximum value in \( V[i..j] \), that is, \( \text{RMQ}_V(i, j) = \arg \max_{k \in [i..j]} V[k] \). We use the following result.

\[ \text{Theorem 2 ([10])}. \] For an integer array \( V \) of length \( n \), there is a data structure with \( 2n + o(n) \) bits of space that supports the RMQs in \( O(1) \) time.

2.3 FM-index

The suffix array \( \text{SA} \) of \( T \) is the integer array of length \( n + 1 \) such that \( \text{SA}[i] \) is the starting position of the lexicographically \( i \)-th suffix of \( T \).\(^1\) We define the string \( L \) (a.k.a. the Burrows-Wheeler Transform (BWT) \([5]\) of \( T \)) of length \( n + 1 \) as follows:

\[
L[i] = \begin{cases} 
8 & (\text{SA}[i] = 1), \\
T[\text{SA}[i] - 1] & (\text{SA}[i] > 1).
\end{cases}
\]

We define the string \( F \) of length \( n + 1 \) as \( F = T[\text{SA}[1]]T[\text{SA}[2]] \cdots T[\text{SA}[n + 1]] \). The so-called LF-mapping \( LF \) is the function defined to map a position \( i \) to \( j \) such that \( \text{SA}[j] = \text{SA}[i] - 1 \) (with the corner case \( LF(i) = 1 \) for \( \text{SA}[i] = 1 \)). A crucial point is that LF-mapping can be efficiently implemented by rank queries on \( L \) and select queries on \( F \) with \( LF(i) = \text{select}_F(\text{rank}_L(i, L[i]), L[i]) \).\(^2\) The occurrences of pattern \( P \) in \( T \) can be answered by finding the maximal interval \([P_a..P_b]\) in the \( \text{SA} \) array such that \( T[\text{SA}[i]..] \) is prefixed by \( P \) iff \( i \in [P_a..P_b] \), and computing the \( \text{SA} \)-values in the interval. For a string \( w \) and character \( c \), the so-called backward search computes the maximal interval in the \( \text{SA} \) prefixed by \( cw \) from that of \( w \) using a similar mechanism of the LF-mapping (see \([7]\) for more details).

\(^1\) Against convention, we include the empty string that starts with the position \( n + 1 \) to \( \text{SA} \). In particular, \( \text{SA}[1] = n + 1 \) holds as the empty string is always the smallest suffix.

\(^2\) In the plain LF-mapping, select queries on \( F \) can be implemented by a simple table that counts, for each character \( c \), the number of occurrences of characters smaller than \( c \) in \( T \), but it is not the case in our generalized LF-mapping for pal-matching.
Table 1 A comparison between \( l_{\text{pal}} \) and \( s_{\text{sp}} \) for \( w = \text{abbbabb} \) and \( w' = bw = \text{babbbabb} \). The values that change when prepending \( b \) to \( w \) are underlined.

| \( w = \) | a | b | b | b | a | b | b |
| \( l_{\text{pal}} \) | 1 | 1 | 2 | 3 | 5 | 3 | 5 |
| \( s_{\text{sp}} \) | \( \infty \) | \( \infty \) | 2 | 2 | 5 | 3 | 2 |
| \( w' = \) | b | a | b | b | a | b | b |
| \( l_{\text{pal}} \) | 1 | 1 | 3 | 2 | 3 | 5 | \( \infty \) |
| \( s_{\text{sp}} \) | \( \infty \) | \( \infty \) | 3 | 2 | 5 | 3 | 2 |

3 Encodings for pal-matching

The pal-matching algorithms in [19] are based on the \( l_{\text{pal}} \)-encoding of a string \( w \), denoted as \( l_{\text{pal}} \). \( l_{\text{pal}} \) is the integer array of length \( |w| \) such that, for any position \( 1 \leq i \leq |w| \), \( l_{\text{pal}}[i] \) is the length of the longest suffix-palindrome of \( w[i..i] \). See Table 1 for example.

Lemma 3 (Lemma 2 in [19]). For any strings \( x \) and \( y \), \( x \) and \( y \) pal-match iff \( l_{\text{pal}}[x] = l_{\text{pal}}[y] \).

Although Lemma 3 is sufficient to design suffix-tree type indexes, it seems that the \( l_{\text{pal}} \)-encoding is not suitable to design FM-index type indexes. For example, more than one position could change when a character is prepended (see Table 1) and this unstable property make messes up lexicographic order of \( l_{\text{pal}} \)-encoded suffixes, which prevents us to implement LF-mapping space efficiently.

In this paper, we introduce a new encoding suitable to design FM-index type indexes for pal-matching. Our new encoding is based on the shortest suffix-palindrome for each prefix, \( s_{\text{sp}} \)-encoding has a stable property when prepending \( a \) character. In what follows, we focus on the opposite direction; \( x \) and \( y \) pal-match iff \( s_{\text{sp}}[x] = s_{\text{sp}}[y] \).

Lemma 4. Two strings \( x \) and \( y \) pal-match iff \( s_{\text{sp}}[x] = s_{\text{sp}}[y] \).

Proof. Since the \( s_{\text{sp}} \)-encoding relies only on palindromic structures, the direction from left to right is clear.

In what follows, we focus on the opposite direction: \( x \) and \( y \) pal-match if \( s_{\text{sp}}[x] = s_{\text{sp}}[y] \). Assume for contrary that \( x \) and \( y \) does not pal-match. Without loss of generality, we can assume that there are positions \( i \) and \( j \) such that \( x[i..j] \) is a palindrome but \( y[i..j] \) is not, with smallest \( j \) if there are many. Note that the smallest assumption on \( j \) implies that \( y[i+1..j-1] \) is a palindrome: If \( y[i+1..j-1] \) is not a palindrome (clearly \( |y[i+1..j-1]| > 1 \) in such a case), \( j-1 \) must be a smaller position that satisfies the above condition because \( x[i+1..j-1] \) is a palindrome. Let \( k = s_{\text{sp}}[j] = s_{\text{sp}}[y][j] \). Since \( x[i..j] \) is a palindrome, it holds that \( 1 < k \leq |x[i..j]| \). Moreover, \( k \neq |y[i..j]| \) as \( y[i..j] \) is not a palindrome. Since the palindrome \( x[i..j] \) has a suffix-palindrome of length \( k \), the prefix \( x[i..i+k-1] \) of length \( k \) is a palindrome, too. On the other hand, since \( y[i..j] \) is not a palindrome that has a suffix-palindrome of length \( k \), the prefix \( y[i..i+k-1] \) of length \( k \) cannot be a palindrome. This contradicts the smallest assumption on \( j \) because \( i+k-1 \) is a smaller position such that \( x[i..i+k-1] \) and \( y[i..i+k-1] \) disagree on their palindromic structures.

In contrast to the \( l_{\text{pal}} \)-encoding, the \( s_{\text{sp}} \)-encoding has a stable property when prepending a character.
Lemma 5. For any string $w$ and character $c$, there is at most one position $i$ ($1 \leq i \leq |w|$) such that $ssp_w[i] \neq ssp_w[i + 1]$. Moreover, if such a position $i$ exists, $ssp_w[i] = \infty$ and $ssp_w[i + 1] = i + 1$.

Proof. By definition it is obvious that $ssp_w[i] = ssp_w[i + 1]$ if $ssp_w[i] \neq \infty$. In what follows, we assume for contrary that there exist two positions $i$ and $i'$ with $1 \leq i < i' \leq |w|$ such that $ssp_w[i] = \infty > ssp_w[i + 1]$ and $ssp_w[i'] = \infty > ssp_w[i' + 1]$. Note that $ssp_w[i + 1] = i + 1$ and $ssp_w[i' + 1] = i' + 1$ by definition, and $(cw)[..i + 1]$ and $(cw)[..i' + 1]$ are palindromes. Since $(cw)[..i + 1]$ is a prefix-palindrome of $(cw)[..i' + 1]$, it is also a suffix-palindrome of $(cw)[..i' + 1]$. It contradicts that $(cw)[..i' + 1]$ is the non-trivial shortest suffix-palindrome of $(cw)[..i' + 1]$.

We consider yet another encoding based on the shortest suffix of $w[..i - 1]$ that is extended outwards when appending a character $w[i]$. The concept is closely related to the $ssp$-encoding because the extended palindrome is the non-trivial shortest suffix-palindrome of $w[..i]$. An advantage of this new encoding is that we can reduce the number of distinct integers to be used to $O(\min(\sigma, \lg |w|))$, which will be used (in a symmetric way) to define $L_{\text{pal}}$ and obtain a space-efficient FM-index specialized for pal-matching.

For any string $w$ we partition the suffix-palindromes (including the empty suffix) by the characters they have immediately to their left and call each group a suffix-pal-group for $w$. We utilize the following lemma.

Lemma 6. For any string $w$, the number of suffix-pal-groups for $w$ is $O(\min(\sigma, \lg |w|))$.

Proof. It is obvious that the number of suffix-pal-groups is at most $\sigma$ because each character is associated to at most one suffix-pal-group. Also it is known that the lengths of the suffix-palindromes can be represented by $O(\lg |w|)$ arithmetic progressions and each arithmetic progression induces a period in the involved suffix (e.g., see [20]). Then we can see that every suffix-palindrome represented by an arithmetic progression is in the same group. Hence there are $O(\lg |w|)$ groups.

The next lemma shows that pal-matching strings share the same structure of suffix-pal-groups.

Lemma 7. Let $x$ and $y$ be strings that pal-match and let $i$ and $j$ be integers with $1 \leq i < j \leq |x| = |y|$. If $x[i+1..]$ and $x[j+1..]$ are palindromes with $x[i] = x[j]$, then $y[i+1..]$ and $y[j+1..]$ are palindromes with $y[i] = y[j]$.

Proof. Since the palindrome $x[i+1..]$ has a suffix-palindrome of length $k = |x[j+1..]|$, it also has a prefix-palindrome of length $k$, that is, $x[i+1..i+k]$ is a palindrome. Also, $x[i+k+1] = x[j]$ holds. Since $x[i+1..i+k+1] = x[i+1..i+k+1]$ is a palindrome.

Since $x$ and $y$ pal-match, $y[i+1..]$, $y[j+1..]$ and $y[i..i+k+1]$ are palindromes. By transition of equivalence induced by the palindromes $y[i..i+k+1]$ and $y[i+1..]$, we can see that $y[i] = y[i+k+1] = y[j]$. Thus the claim holds.

Let the shortest palindrome in a suffix-pal-group be the representative of the group. We assign consecutive integer identifiers starting from 1 to the suffix-pal-groups in increasing order of their representative’s lengths. See Figure 2 for example.

For any string $w$, we define the shortest suffix-pal-group encoding $sspg_w$ of $w$ as the integer array of length $|w|$ such that, for any position $1 \leq i \leq |w|$, $sspg_w[i]$ is the identifier assigned to the suffix-pal-group of the suffix-palindrome in $w[..i-1]$ that is extended outwards by appending $w[i]$, if such exists, and otherwise $\infty$. See Table 2 and Figure 3 for example. Since
Figure 2 An example of suffix-pal-groups for \texttt{bababacababacababacababa}. The number enclosed in a circle denotes the pal-group-id. The suffix-palindromes in the suffix-pal-group with identifier 1 (resp. 2 and 3) have a (resp. b and c) immediately to their left. The identifiers are given in increasing order of their representative’s lengths, that is, \(|\varepsilon| = 0, |a| = 1\) and \(|ababa| = 5\).

the non-trivial shortest suffix of \(w[\ldots i]\) is extended outwards from the representative of the suffix-pal-group for \(w[1..i-1]\) that has \(w[i]\) immediately to the left, \(\text{sspg}_w[i]\) has essentially equivalent information to \(\text{ssp}_w[i]\). Formally the next lemma holds.

\begin{lemma}
For any string \(x\) of length \(k\), suppose we have the set of lengths of the representatives of suffix-pal-groups of \(x[1..k-1]\). Given \(\text{sspg}_x[k]\) we can identify \(\text{ssp}_x[k]\), and vice versa.
\end{lemma}

\begin{proof}
It is clear that \(\text{ssp}_x[k] = \infty\) iff \(\text{sspg}_x[k] = \infty\). Given \(\text{sspg}_x[k] \neq \infty\) we can identify \(\text{ssp}_x[k]\) from the representative of the suffix-pal-group with identifier \(\text{sspg}_x[k]\). Given \(\text{ssp}_x[k] \neq \infty\) we can identify \(\text{sspg}_x[k]\) from the representative that has length \(\text{ssp}_x[k] - 2\).
\end{proof}

The next lemma shows that the \(\text{sspg}\)-encoding is another encoding for pal-matching, and induces the same lexicographic order with the \(\text{ssp}\)-encoding.

\begin{lemma}
Let \(x\) and \(y\) be strings of length \(k\) such that \(\text{ssp}_x[1..k-1] = \text{ssp}_y[1..k-1]\). Then, \(\text{ssp}_x[k] = \text{ssp}_y[k]\) iff \(\text{sspg}_x[k] = \text{sspg}_y[k]\). Also, \(\text{ssp}_x[k] < \text{ssp}_y[k]\) iff \(\text{sspg}_x[k] < \text{sspg}_y[k]\).
\end{lemma}

\begin{proof}
It follows from Lemma 7 that \(x[1..k-1]\) and \(y[1..k-1]\) have the same structure of suffix-pal-groups. By Lemma 8, \(\text{ssp}_x[k] = \text{ssp}_y[k]\) if \(\text{sspg}_x[k] = \text{sspg}_y[k]\), and vice versa. Since the identifiers of suffix-pal-groups are given in increasing order of their representative’s lengths, it holds that \(\text{ssp}_x[k] < \text{ssp}_y[k]\) if and only if \(\text{sspg}_x[k] < \text{sspg}_y[k]\).
\end{proof}

For any string \(w\), let \(\pi(w) = \text{sspg}_{w,n}[|w|]\). Intuitively, \(\pi(w)\) holds the information from which prefix-palindrome of \(w[2..]\) the non-trivial shortest prefix-palindrome of \(w\) is extended, and the information is encoded with the identifier defined in the completely symmetric way as the case of the suffix-pal-groups. The function \(\pi(\cdot)\) will be applied to the suffixes of \(T\) to define \(F_{\text{pal}}\) and \(L_{\text{pal}}\), and the next lemma is a key to implement LF-mapping for our PalFM-index.
Table 2 A comparison between \( \text{ssp}_w \) and \( \text{sspg}_w \) for \( w = \text{babbbabb} \). \( \text{ssp}_w[6] = 5 \) because the non-trivial shortest suffix-palindrome of \( w[1..6] = \text{babbb} \) is \( \text{abbba} \), which is of length 5. On the other hand, \( \text{sspg}_w[6] = 2 \) because the shortest suffix-palindrome \( \text{abbba} \) ending at 6 is extended from \( \text{bbb} \) and the suffix-pal-group to which \( \text{bbb} \) belongs for \( w[1..5] = \text{babbb} \) has the identifier 2.

\[ w = \text{babbbabb} \]

\[
\begin{array}{cccccccc}
\text{ssp}_w & = & \infty & \infty & 3 & 2 & 2 & 5 & 3 & 2 \\
\text{sspg}_w & = & \infty & \infty & 2 & 1 & 1 & 2 & 2 & 2 \\
\end{array}
\]

\[ w = \text{babbbabb} \]

\[ \text{sssp}_w[6] = 2 \]

Figure 3 Illustration to show \( \text{sspg}_w[6] = 2 \) for \( w = \text{babbbabb} \).

Lemma 10. Let \( x \) and \( y \) be strings of length \( \geq 1 \) such that \( \pi(x) = \pi(y) \). Then, \( \text{ssp}_x \prec \text{ssp}_y \) iff \( \text{ssp}_x[2..i] \prec \text{ssp}_y[2..i] \).

Proof. Let \( i \) be the largest integer such that \( x[2..i] \) and \( y[2..i] \) pal-match. Since \( \pi(x) = \pi(y) \), using Lemma 9 in a symmetric way, it holds that \( x[2..i] \) and \( y[2..i] \) pal-match. Recall Lemma 5 that at most one \( \infty \) in \( \text{ssp}_x[2..i] \) (resp. \( \text{ssp}_y[2..i] \)) turns into the largest possible integer at the changed position when prepending \( x[1] \) (resp. \( y[1] \)). We analyze the cases focusing on the changed positions:

1. The claim clearly holds if neither \( \text{ssp}_x \) nor \( \text{ssp}_y \) has the changed position less than or equal to \( i + 1 \).

2. If both of \( \text{ssp}_x \) and \( \text{ssp}_y \) have the changed position at \( j \leq i + 1 \), it holds that \( \text{ssp}_x[j] = \text{ssp}_y[j] = j \) and \( \text{ssp}_x[2..i][j - 1] = \text{ssp}_y[2..i][j - 1] = \infty \), which also indicates that \( j < i + 1 \). Since this change does not affect the lexicographic order, the claim holds. See the left part of Figure 4 for an illustration of this case.

3. Assume \( \text{ssp}_y \) has the changed position at \( j \leq i + 1 \), but \( \text{ssp}_x \) does not. Since \( x[2..i] \) and \( y[2..i] \) pal-match, \( j \) cannot be less than \( i + 1 \), and hence, \( j = i + 1 \) and \( \text{ssp}_y[i + 1] = \text{ssp}_x[2..i][i] \). Note that the lexicographic order between \( \text{ssp}_x \) and \( \text{ssp}_y \) (resp. \( \text{ssp}_x[2..i] \) and \( \text{ssp}_y[2..i] \)) is determined by that between \( \text{ssp}_x[i + 1] \) and \( \text{ssp}_y[i + 1] \) (resp. \( \text{ssp}_x[2..i][i] \) and \( \text{ssp}_y[2..i][i] \)). Since the lexicographic order between \( \text{ssp}_x[i + 1] \) and \( \text{ssp}_y[i + 1] \) is the same as that between \( \text{ssp}_x[2..i][i] \) and \( \text{ssp}_y[2..i][i] \), the claim holds. See the right part of Figure 4 for an illustration of this case.

Thus, we conclude that the lemma holds.
4 Computational results for new encodings

In this section, we show that the ssp- and sspg-encodings can be computed in linear time for a given string.

We use the following known results.

▶ Lemma 11 ([26]). For any string \( w \), we can compute all the maximal palindromes in \( O(|w|) \) time.

▶ Lemma 12 (Lemma 3 in [19]). For any string \( w \), we can compute \( \text{lpal}_w \) in \( O(|w|) \) time.

Using Lemmas 11 and 12, we obtain:

▶ Lemma 13. For any string \( w \), we can compute \( \text{ssp}_w \) in \( O(|w|) \) time.

Proof. Manacher’s algorithm [26] can compute the radius of the maximal palindrome in increasing order of centers in linear time. It can be extended to compute the length \( \text{lpal}_w[i] \) of the longest palindrome ending at each position \( i \) because the maximal palindrome with the smallest center that ends at position \( i \) gives us the longest suffix-palindrome ending at \( i \) by truncating the palindrome at \( i \) (e.g., see Lemma 3 of [19]). In a similar way, we can compute the length \( \text{lpal}'_w[i] \) of the second longest palindrome ending at \( i \).

Using \( \text{lpal}_w \) and \( \text{lpal}'_w \), we can compute \( \text{ssp}_w[i] \) in increasing order as follows:

1. If \( \text{lpal}_w[i] = 1 \), then \( \text{ssp}_w[i] = \infty \).
2. If \( \text{lpal}_w[i] > 1 \) and \( \text{lpal}'_w[i] = 1 \), then \( \text{ssp}_w[i] = \text{lpal}_w[i] \).
3. If \( \text{lpal}_w[i] > 1 \) and \( \text{lpal}'_w[i] > 1 \), then \( \text{ssp}_w[i] = \text{ssp}_w[i - \text{lpal}_w[i]] + \text{lpal}'_w[i] \).

In the third case, we use the fact that the non-trivial shortest suffix-palindrome ending at \( i \) has length \( \leq \text{lpal}'_w[i] \) and it ends at \( i - \text{lpal}_w[i] + \text{lpal}'_w[i] \), too.

Clearly all can be done in \( O(|w|) \) time. ▶

For any string \( w \), let \( G_w \) denote the array of length \( |w| \) such that \( G_w[i] \) stores the number of suffix-pal-groups for \( w[..i] \).

▶ Lemma 14. For any string \( w \), we can compute \( G_w \) in \( O(|w|) \) time.
The PalFM-index of $T$ conceptually sort the suffixes of $T$ in lexicographic order of their ssp-encodings (or equivalently sspg-encodings). Let $\text{SA}_\text{pal}$ be the integer array of length $n + 1$ such that $\text{SA}_\text{pal}[i]$ is the starting position of the $i$-th suffix of $T$ in ssp-encoded order. We define the strings $F_\text{pal}$ and $l_\text{pal}$ of length $n + 1$ based on $\pi$ function applied to the sorted suffixes. Formally, for any position $i$ ($1 \leq i \leq n + 1$) we define:
Thus we focus on a single step of backward search. In a general setting, for any string \( T \) with \( n \) characters over an alphabet of size \( \sigma \), we show how to compute \( \pi \) of prefix-pal-groups of \( T \) in \( O(n) \) time.

\[
\pi[i] = \begin{cases} 
\emptyset & \text{if } i = 1, \\
\pi(T[\text{SA}_\text{pal}[i]..]) & \text{otherwise.}
\end{cases}
\]

\[
L_\text{pal}[i] = \begin{cases} 
\emptyset & \text{if } \text{SA}_\text{pal}[i] = 1, \\
\pi(T[\text{SA}_\text{pal}[i] - 1..]) & \text{otherwise.}
\end{cases}
\]

See Figure 6 for an example.

As in the case of LF, we define a function \( LF_\text{pal} : i \mapsto j \) so that \( \text{SA}_\text{pal}[j] = \text{SA}_\text{pal}[i] - 1 \) (with the corner case \( LF_\text{pal}(i) = 1 \) for \( \text{SA}_\text{pal}[i] = 1 \)). Thanks to Lemma 10, for any value \( c \), the suffixes used to obtain \( i \)-th \( k \) in \( L_\text{pal} \) and in \( F_\text{pal} \) are the same, which enables us to implement the \( LF_\text{pal} \) function by \( LF_\text{pal}(i) = \text{select}_{F_\text{pal}}(\text{rank}_{L_\text{pal}}(i, 0), L_\text{pal}[i]) \). See Figure 7 for an illustration.

For any string \( w \), let \( w \)-interval refer to the maximal interval \([b, e]\) such that \( \text{ssP}_{T[\text{SA}_\text{pal}[i]..]} \) is prefixed by \( \text{ssP}_w \), where \( w \)-interval is empty if there is no substring of \( T \) that pal-matches with \( w \). Notice that the substring of \( T \) of length \( |w| \) starting at \( \text{SA}_\text{pal}[i] \) pal-matches with \( w \) if and only if \( i \in [b, e] \). A single step of backward search computes \( cw \)-interval from \( w \)-interval for some character \( c \).

The following theorems are the main contributions of this paper.

**Theorem 16.** Let \( T \) be a string of length \( n \) over an alphabet of size \( \sigma \). There is a data structure of \( 2n \log \min(\sigma, \log n) + 2n + o(n) \) bits of space to support the counting queries for the pal-matching problem in \( O(m) \) time, where \( m \) is the length of a given pattern \( P \).

**Proof.** We use the data structures of Theorem 1 for \( L_\text{pal} \) and \( F_\text{pal} \), and the \( \text{RMQ} \) data structure of Theorem 2 for the integer array \( V \) with \( V[i] = LF_\text{pal}(i) \). Since the number of distinct symbols in \( L_\text{pal} \) and \( F_\text{pal} \) are \( O(\min(\sigma, \log n)) \) by Lemma 6, the data structures occupy \( 2n \log \min(\sigma, \log n) + 2n + o(n) \) bits of space in total and all queries (\( \text{rank} \), \( \text{select} \), \( \text{rangeCount} \) and \( \text{RMQ} \)) can be supported in \( O(1) \) time.

The number of occurrences of \( P \) can be answered by computing the width of \( P \)-interval. Thus we focus on a single step of backward search. In a general setting, for any string \( w \) and a character \( c \), we show how to compute \( cw \)-interval \([b..e]\) in \( O(1) \) time from \( w \)-interval \([b..e]\), \( \pi(cw) \) and the number \( g \) of prefix-pal-groups of \( w \). The procedure differs depending on \( \pi(cw) = \infty \) or not.

| \( i \) | \( T[i..] \) | \( \text{ssP}_{T[i..]} \) | \( \text{ssP}_{T[\text{SA}_\text{pal}[i]..]} \) | \( \text{SA}_\text{pal}[i] \) | \( F_\text{pal}[i] \) | \( L_\text{pal}[i] \) | \( LF_\text{pal}(i) \) |
|-------|-------|-----------------|-----------------|--------|--------|--------|--------|
| 1     | ababbcbc | \( \infty:2432:33 \) | \( \varepsilon \) | 10     | \$      | \infty  | 2      |
| 2     | bbabbcbc | \( \infty:32:33 \) | \( \infty \)    | 9      | \infty  | \infty  | 5      |
| 3     | abbbbc | \( \infty:32:33 \) | \( \infty:2:33 \) | 2      | 1       | 1       | 6      |
| 4     | abbbbc | \( \infty:2:33 \) | \( \infty \)    | 5      | 1       | \infty  | 7      |
| 5     | bbabc | \( \infty:2:33 \) | \( \infty \)    | 8      | \infty  | 2       | 8      |
| 6     | babc | \( \infty:33 \) | \( \infty:2432:33 \) | 1      | 2       | 8       | 1      |
| 7     | cbc | \( \infty:3 \) | \( \infty:2:33 \) | 4      | \infty  | 2       | 9      |
| 8     | bc | \( \infty:3 \) | \( \infty:3 \)   | 7      | 2       | 2       | 10     |
| 9     | c | \( \infty \) | \( \infty:32:33 \) | 3      | 2       | 1       | 3      |
| 10    | \( \varepsilon \) | \( \varepsilon \) | \( \infty:33 \) | 6      | 2       | 1       | 4      |
1. When \( \pi(cw) = k \neq \infty \). Using Lemma 9 in a symmetric way, \([b',e']\) is obtained by mapping the positions of \( \pi(cw) \) in \( L_{pal}[b,e] \) by the \( L_{F_{pal}} \) function. More specifically, \( b' = select_{F_{pal}}(rank_{L_{pal}}(b - 1, k) + 1, k) \) and \( e' = select_{F_{pal}}(rank_{L_{pal}}(e, k), k) \), which can be computed in \( O(1) \) time.

2. When \( \pi(cw) = \infty \). We note that \([b',e']\) is the maximal interval such that \( T[SA_{pal}[i]..] \) does not have non-trivial prefix-palindrome (i.e. \( \pi(T[SA_{pal}[i]..]) = \infty \)) or \( T[SA_{pal}[i]..] \) has the non-trivial shortest prefix-palindrome of length longer than \( |cw| \) (i.e. \( \pi(T[SA_{pal}[i]..]) > g \)). Thus, \( e' = b' + 1 \) is equivalent to the number of occurrences of values larger than \( g \) in \( L_{pal}[b,e] \), which can be computed in \( rangeCount_{L_{pal}}(b, e, g, \infty) \) in \( O(1) \) time. Moreover, it holds that \( e' = LF_{pal}(RMQ_{\pi}(b,e)) \) because \( ssp(T[SA_{pal}[i]..]) \) with \( \pi(T[SA_{pal}[i]..]) = L_{pal}[i] > g \) is always lexicographically larger than \( ssp(T[SA_{pal}[j]..]) \) with \( \pi(T[SA_{pal}[j]..]) = L_{pal}[j] \leq g \). Thus, we can compute \([b',e']\) in \( O(1) \) time.

Backward search for \( P \) requires \( \pi(P[i..]) \) and the number \( g \) of prefix-pal-groups of \( P[i..] \) for all \( 1 \leq i \leq m \), which can be computed by \( sspg_{PN} \) and \( G_{PN} \) in \( O(m) \) time using Lemmas 15 and 14.

Putting all together, we get the theorem.

**Theorem 17.** Let \( T \) be a string of length \( n \) over an alphabet of size \( \sigma \) and \( \Delta \) be an integer in \([1,n]\). There is a data structure of \( 2n \lg \min(\sigma, \lg n) + \frac{n}{\Delta} \lg n + 3n + o(n) \) bits of space to support the locating queries for the pal-matching problem in \( O(m + \Delta \text{occ}) \) time, where \( m \) is the length of a given pattern \( P \) and \( \text{occ} \) is the number of occurrences to report.
Proof. We use the data structure and the algorithm of Theorem 16 to compute $P$-interval in $2n(1 + \lg \min(\sigma, \lg n)) + o(n)$ bits of space and $O(m)$ time. The occurrences of $P$ (in the sense of pal-matching) can be answered by the $SA_{pal}$-values in $P$-interval. We employ exactly the same sampling technique used in the FM-index to retrieve $SA$-values (e.g., see [7]): We make a bit vector $B$ of length $n + 1$ marking the positions $i$ in $SA_{pal}$ such that $SA_{pal}[i] = \Delta k + 1$ for some integer $k$, and the sparse suffix array $S$ holding only the marked $SA_{pal}$-values in the order. $B$ is equipped with a data structure to support the rank queries and the additional space to Theorem 16 is $\frac{n}{\Delta} \lg n + n + o(n)$ bits in total.

If position $i$ is marked, $SA_{pal}[i]$ is retrieved by $S[\text{rank}_B(i, 1)]$ in $O(1)$ time. If position $i$ is not marked, we apply LF-mapping $k$ times from $i$ until we reach a marked position $j$ and retrieve $SA_{pal}[i]$ by $S[\text{rank}_B(j, 1)] + k$. Since text positions are marked every $\Delta$ positions, the number $k$ of LF-mapping steps is at most $\Delta$, and hence, $SA_{pal}[i]$ can be retrieved in $O(\Delta)$ time. Therefore we can report each occurrence of $P$ in $O(\Delta)$ time, and the theorem follows.

6 Conclusions and future work

In this paper, we developed new encoding schemes for pal-matching and proposed the PalFM-index, a space-efficient index for pal-matching based on the FM-index. Future work includes to present an efficient construction algorithm of the PalFM-index, and to reduce the space requirement (e.g. by incorporating with the idea of [13]). Another interesting research direction would be to develop a general framework to design FM-index type indexes in generalized pattern matching. We believe that switching encoding from $lpal$ to $ssp$ to design the PalFM-indexes gives a good hint to pursue this direction, and conjecture that any generalized pattern matching under a substring consistent equivalent relation [27] admits such shortest positional encodings to design FM-index type indexes.

References

1. Jean-Paul Allouche, Michael Baake, Julien Cassaigne, and David Damanik. Palindrome complexity. *Theor. Comput. Sci.*, 292(1):9–31, 2003.
2. Mira-Cristiana Anisiu, Valeriu Anisiu, and Zoltán Kása. Total palindrome complexity of finite words. *Discrete Mathematics*, 310(1):109–114, 2010. doi:10.1016/j.disc.2009.08.002.
3. Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM) 2017*, pages 23:1–23:12, 2017. doi:10.4230/LIPIcs.CPM.2017.23.
4. Srecko Brlek, Sylvie Hamel, Maurice Nivat, and Christophe Reutenauer. On the palindromic complexity of infinite words. *Int. J. Found. Comput. Sci.*, 15(2):293–306, 2004. doi:10.1142/S012905410400242X.
5. Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994.
6. Xavier Droubay, Jacques Justin, and Giuseppe Pirillo. Episturmian words and some constructions of de Luca and Rauzy. *Theor. Comput. Sci.*, 255(1-2):539–553, 2001. doi:10.1016/S0304-3975(99)00320-5.
7. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.
8. Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.
9. Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48, 2014. StringMasters 2012 & 2013 Special Issue (Volume 1). doi:10.1016/j.jda.2014.08.001.
PalFM-Index: FM-Index for Palindrome Pattern Matching

10 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
11 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In *Proc. 25th Annual European Symposium on Algorithms (ESA) 2017*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.
12 Zvi Galil and Joel I. Seiferas. A linear-time on-line recognition algorithm for “palstar”. *J. ACM*, 25(1):102–111, 1978. doi:10.1145/322047.322056.
13 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2017*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.
14 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In *Proc. 28th International Symposium on Algorithms and Computation (ISAAC) 2017*, pages 38:1–38:15, 2017. doi:10.1007/978-3-662-54542-5_35.
15 Amy Glen, Jacques Justin, Steve Widmer, and Luca Q. Zamboni. Palindromic richness. *Eur. J. Comb.*, 30(2):510–531, 2009. doi:10.1016/j.ejc.2008.04.006.
16 Alexander Golynski, Rajeev Raman, and S. Srinivasa Rao. On the redundancy of succinct data structures. In Joachim Gudmundsson, editor, *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT) 2008*, volume 5124 of *Lecture Notes in Computer Science*, pages 148–159. Springer, 2008.
17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2003*, pages 841–850. ACM/SIAM, 2003.
18 Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Counting and verifying maximal palindromes. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE) 2010*, pages 135–146, 2010.
19 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome pattern matching. *Theor. Comput. Sci.*, 483:162–170, 2013. doi:10.1016/j.tcs.2012.01.047.
20 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM) 2014*, volume 8486 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2014.
21 Ignacio Tinoco Jr., Olke C. Uhlenbeck, and Mark D. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971.
22 Sung-Hwan Kim and Hwan-Gue Cho. A compact index for cartesian tree matching. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM) 2021*, volume 191 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
23 Sung-Hwan Kim and Hwan-Gue Cho. Simpler FM-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. doi:10.1016/j.ipl.2020.106026.
24 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
25 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Pal $k$ is linear recognizable online. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, pages 289–301, 2015. doi:10.1007/978-3-662-46078-8_24.
26 Glenn K. Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.
27 Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theor. Comput. Sci.*, 656:225–233, 2016.
28 Antonio Restivo and Giovanna Rosone. Burrows-wheeler transform and palindromic richness. *Theor. Comput. Sci.*, 410(30-32):3018–3026, 2009. doi:10.1016/j.tcs.2009.03.008.

29 Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018. doi:10.1016/j.ejc.2017.07.021.