Efficient Index for Square Pattern Matching

Po-Chun Chen
National Tsing Hua University, Hsinchu, Taiwan

Che-Wei Tsao
National Tsing Hua University, Hsinchu, Taiwan

Wing-Kai Hon
National Tsing Hua University, Hsinchu, Taiwan

Dominik Köppl
University of Yamanashi, Kofu, Japan

Abstract

A string S is called a *square* if it can be written as the concatenation of two identical strings. Two strings P and Q of the same length are said to *square match* if, for every substring of P, it is a square if and only if the corresponding substring of Q is also a square. The *square pattern matching* problem asks for locating all substrings of a given text T of length n that square match a query pattern P of length m. This notion captures similarity in repetition structures and is motivated by applications in areas such as bioinformatics and music structure analysis.

In this paper, we introduce a novel technique, called the *longest prefix square* (LPS) encoding, which represents the square structure of a string as an integer array of the same length. We show that two strings square match if and only if they have identical LPS encodings. Based on this result, we construct an index solving the square pattern matching problem in time $O(m \lg m + occ)$ using $O(n \lg^2 n)$ bits of space, where occ denotes the number of occurrences of substrings in T that square match P. If the LPS encoding of P is precomputed, the query time improves to $O(m + occ)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, pattern matching, indexing, squares

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.35

Funding *Wing-Kai Hon*: This work was supported by Taiwan NSTC Grant Number 113-2221-E-007-139.
Dominik Köppl: This work was supported by JSPS KAKENHI Grant Number 25K21150.

Acknowledgements We thank the anonymous reviewers for their comments, which greatly strengthen our results.

1 Introduction

A string S is called a square if it can be expressed as the concatenation of two identical non-empty strings, that is, $S = WW$. As a fundamental unit of repetition, squares have attracted considerable attention for decades, particularly in areas such as bioinformatics [14] and music structure analysis [20, 21]. Most existing studies focus on exact or approximate pattern matching, where the target substrings are required to be identical to, or differ slightly from, a given query pattern.

In this paper, we propose a new type of matching problem, called the *square pattern matching* problem. We say that two strings U and V of the same length are *square matched* if, for every substring of U, it is a square if and only if the corresponding substring of V is also a square. This notion of square match captures the idea that the two strings share identical repetition structures (or, equivalently, identical run structures [3]). The *Square Pattern Matching* problem is to locate all substrings of a given text T that square match a query pattern P.

Several studies on structural matching [1] have been proposed, including parameterized pattern matching [2], order-preserving matching [11, 8], Cartesian tree matching [18, 22], and substring consistent equivalence relation (SCER) matching [15, 9]. Recently, palindrome pattern matching [10, 17] has also been introduced. Our problem is inspired by this line of research and can be viewed as a structural counterpart in which the underlying property is repetition rather than palindromic symmetry.

Our core idea is to introduce a novel technique, called the *longest prefix square* (LPS) encoding, which encodes the repetition structure of a string into a sequence of integers of the same length. Using this encoding, we can verify whether two strings are square matched by simply comparing their LPS encodings. We further introduce the LPS index to locate all substrings of a given text T of length n that square match with a query pattern P of length m. The query can be answered in $O(m \lg m + occ)$ time using $O(n \lg^2 n)$ bits of space, where occ denotes the number of occurrences of substrings in T that square match P. If the LPS encoding of P is precomputed, the query time improves to $O(m + occ)$.

Our contributions can be summarized as follows:

- **LPS Encoding:** We introduce the *longest prefix square* (LPS) encoding, which characterizes the square structure of a string by an integer array of the same length. We prove that two strings square match if and only if they have identical LPS encodings.
- **LPS Index Construction:** We design algorithms to construct the LPS index, consisting of the LPS table and the diagonal trie. For a text T of length n, the LPS index can be constructed in $O(n^2)$ preprocessing time.
- **LPS Index Compression:** We compress the LPS index for a text of length n from a naïve $n^2 \lg n$ bit representation to $O(n \lg^2 n)$ bits, while preserving query efficiency.

The organization of the paper is as follows. Section 2 presents preliminary definitions. In Section 3, we introduce our main contribution, the LPS encoding and the LPS index, for solving the square pattern matching problem. Section 4 describes how to compress and construct the LPS index. Finally, Section 5 concludes the paper and discusses possible directions for future work.

2 Preliminaries

We consider a string $S = S[1]S[2] \dots S[n]$ of length n, where each character is drawn from an alphabet of constant size σ. For any $1 \le i \le j \le n$, we denote by $S[i..j]$ the substring $S[i]S[i+1] \dots S[j]$. If $i > j$, we define $S[i..j]$ to be the empty string.

The concatenation of two strings U and V is denoted by UV. If a string S can be written as $S = UV$, where U and V are (possibly empty) strings, then U and V are called a *prefix* and a *suffix* of S, respectively. A *proper prefix* (resp., *proper suffix*) of S is a prefix (resp., suffix) of S that is not equal to S itself.

For a string S and an integer $t \ge 1$, we denote by S^t the string obtained by concatenating S with itself t times. If S can be written as $S = W^k$ for a non-empty string W and an integer $k \ge 2$, then S is called *non-primitive*; otherwise, it is *primitive*.

2.1 Periodicity

If a string S can be written as $S = R^k R'$, where $k \ge 2$, R is a non-empty string, and R' is a prefix of R, then $|R|$ is called a period of S. The shortest such string R is called the *root* of S.

The following is an important result on the periods of strings:

▶ **Theorem 1** (Fine-Wilf Theorem [5]). *Let S be a string, and suppose that x and y are periods of S. If $|S| \geq x + y - \gcd(x, y)$, then $\gcd(x, y)$ is also a period of S.*

2.2 Squares

A string S is called a *square* if it can be written as $S = WW$ for a non-empty string W. A square W^2 is called a *primitive square* if W is primitive; otherwise, it is a *non-primitive square*.

The following is an important result on primitive squares:

▶ **Lemma 2** (Three Squares Lemma [4]). *Let U^2, V^2, W^2 be three prefixes of a string S such that U, V, W are primitive and $|W| < |V| < |U|$. Then $|V| + |W| < |U|$.*

▶ **Corollary 3.** *A string S of length n has at most $O(\lg n)$ prefixes that are primitive squares.*

Proof. When we arrange the primitive square prefixes of S in increasing order of their lengths, their lengths must at least double for every two such prefixes (by Lemma 2). ◀

2.3 Run structure

A *run* [13] in a string S is represented by a tuple (s, t, p), where $S[s..t] = R^k R'$ for a root R of length p, an integer $k \geq 2$, and R' is a prefix of R; furthermore, the interval $S[s..t]$ needs to be maximal (where it cannot be extended to the left or to the right), so that $S[s-1] \neq S[s+p-1]$ and $S[t+1] \neq S[t-p+1]$. For example, `aabaabac` has three runs: $(1, 2, 1)$, $(1, 7, 3)$, and $(4, 5, 1)$, as shown in Figure 1.

The maximum number of runs in a string S of length n is $O(n)$ [13] and can be identified in $O(n)$ time for ordered alphabets [6].

We say that a position $1 \leq i \leq n$ is *covered* by a run (s, t, p) if and only if $s \leq i \leq t-2p+1$. It is easy to see that a square starts at position i if and only if i is covered by a run. Thus, we obtain the following lemma.

▶ **Lemma 4.** *Let S be a string of length n. Each position i (with $1 \leq i \leq n$) is covered by at most $O(\lg n)$ runs.*

Proof. Assume that a position i is covered by more than $O(\lg n)$ runs. Then there are more than $O(\lg n)$ primitive squares starting at i, contradicting Corollary 3. ◀

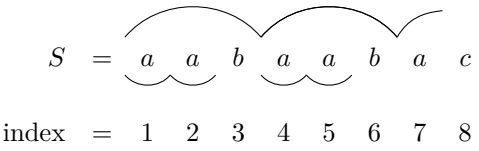

Figure 1 The run structure of `aabaabac`.

2.4 Model of Computation

Throughout this paper, we assume that the word size w is large enough to hold the input size encoded in binary. In other words, when dealing with strings of length n, $w = \Theta(\lg n)$ will be sufficient. Furthermore, we assume the unit-cost word RAM model [7] as the model of computation, in which standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time.

3 Longest Prefix Square (LPS) Index

In this section, we first formally define the problem under consideration. We then introduce several theorems and the LPS index for solving the problem. We begin by introducing a measure for verifying whether two strings have the same square structure, which is defined as follows.

▶ **Definition 5** (Square Match). *Let X and Y be two strings of the same length. We say that X and Y are* square matched, *if for every pair of positions $i \leq j$, the substring $X[i..j]$ is a square if and only if $Y[i..j]$ is a square.*

Given a string S and a query pattern P, our goal is to locate all substrings of S that are square matched with P. We define the set of such locations as follows.

▶ **Definition 6** ($Loc_{sqr}(S, P)$). *Given a string S of length n and a query pattern P of length m, $Loc_{sqr}(S, P)$ is the set of the positions i of S such that $S[i..i + m - 1]$ and P are square matched, where $1 \leq i \leq n - m + 1$.*

We next describe how to verify whether two strings of length n are square matched. A straightforward brute-force approach runs in $O(n^3)$ time: there are $O(n^2)$ substrings, and for each substring one can check whether it is a square in $O(n)$ time using the KMP algorithm [12]. To reduce the time complexity, we introduce the LPS encoding technique.

▶ **Definition 7** (Longest Prefix Square (LPS)). *Let S be a string of length n. The* longest prefix square *(LPS) of S is the longest prefix of S that is a square. Formally, let*

$$\ell = \max\{0\} \cup \{k \mid 1 \leq k \leq n \text{ and } S[1..k] \text{ is a square}\}.$$

If $\ell > 0$, the LPS of S is $S[1..\ell]$; otherwise, the LPS is defined to be the empty string.

By computing the LPS length of each suffix of S, we obtain the LPS array, defined as follows.

▶ **Definition 8** (LPS Array). *Let S be a string of length n. The LPS array of S, denoted by $\texttt{LPS}_S[1..n]$, is defined such that for each $1 \leq i \leq n$, $\texttt{LPS}_S[i]$ equals the length of the LPS of the suffix $S[i..n]$.*

An example is shown in Figure 2(a). The array $\texttt{LPS}_S$ can be computed using Algorithm 1. Step 1 takes $O(n)$ time [6]. Step 2 can be performed in $O(n)$ time using bucket sort. In Step 3, each iteration processes at most $O(\lg n)$ runs by Lemma 4. Moreover, the runs covering all positions can be computed in $O(n)$ total time using a sweep-line algorithm. Therefore, Step 3 takes $O(n \lg n)$ time in total, and $\texttt{LPS}_S$ can be computed in $O(n \lg n)$ time.

Algorithm 1 Algorithm for calculating $\texttt{LPS}_S$.

Given a string S of length n:

1. Compute the *run structure* of S.
2. Sort all runs of S in lexicographic order.
3. For $i = 1$ to n do:
   - $v \leftarrow 0$;
   - For all runs $r_j = (s_j, t_j, p_j)$ that cover position i do:
     - $v \leftarrow \max\left(v, \left\lfloor \frac{t_j - i + 1}{2p_j} \right\rfloor \cdot 2p_j\right)$;
   - $\texttt{LPS}_S[i] \leftarrow v$;

(a)

T	=	a	b	b	a	b	b
$\texttt{LPS}_T$	=	6	2	0	0	2	0

(b)

T	a	b	b	a	b	b
$\ell = 1$	0	0	0	0	0	0
$\ell = 2$	0	2	0	0	2	0
$\ell = 3$	0	2	0	0	2	0
$\ell = 4$	0	2	0	0	2	0
$\ell = 5$	0	2	0	0	2	0
$\ell = 6$	6	2	0	0	2	0

(c)

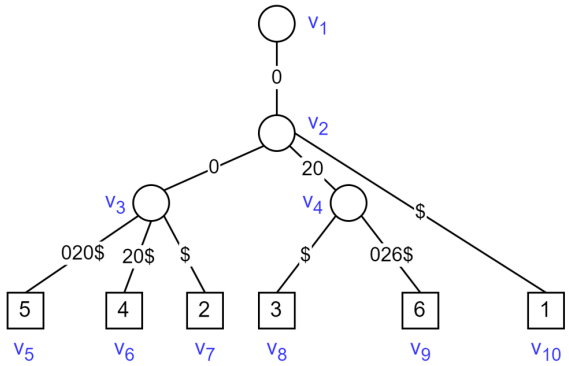

Figure 2 Example illustrating (a) the LPS array, (b) the LPS table, and (c) the diagonal trie of the string `abbabb`.

The LPS array allows us to verify whether two strings are square matched by simply checking whether their LPS arrays are identical, as formalized in the following theorem.

▶ **Theorem 9.** *The two strings X and Y are square matched if and only if* $\texttt{LPS}_X = \texttt{LPS}_Y$.

Proof. ($\Rightarrow$) Suppose that X and Y are square matched. By Definition 5, for every position i and every $\ell \geq 0$, the substring $X[i..i + 2\ell - 1]$ is a square if and only if $Y[i..i + 2\ell - 1]$ is a square. Hence, for each position i, the maximum such ℓ is the same for X and Y, implying that $\texttt{LPS}_X[i] = \texttt{LPS}_Y[i]$. Therefore, $\texttt{LPS}_X = \texttt{LPS}_Y$.

($\Leftarrow$) We prove this direction by induction on the string length n. Assume that $\texttt{LPS}_X = \texttt{LPS}_Y$. For the base cases, when $n = 1$, no square exists and $\texttt{LPS}_X = \texttt{LPS}_Y = \langle 0 \rangle$. When $n = 2$, either $\texttt{LPS}_X = \texttt{LPS}_Y = \langle 0, 0 \rangle$ or $\texttt{LPS}_X = \texttt{LPS}_Y = \langle 2, 0 \rangle$. In the former case, both X and Y consist of two distinct characters; in the latter case, both consist of two identical characters. In either case, X and Y are square matched.

Assume that the statement holds for all $n < k$, and consider the case $n = k$. Let X' and Y' denote the strings obtained from X and Y by removing their first character, respectively. Since the LPS array depends only on the suffixes starting at different positions, the last $k - 1$ entries of $\texttt{LPS}_X$ coincide with those of $\texttt{LPS}_{X'}$, and similarly for Y. Therefore, we have $\texttt{LPS}_{X'} = \texttt{LPS}_{Y'}$. By the induction hypothesis, X' and Y' are square matched, which implies that $X[2..n]$ and $Y[2..n]$ are square matched.

It remains to show that, for every $1 \leq h \leq n$, the substring $X[1..h]$ is a square if and only if $Y[1..h]$ is a square. Let $\texttt{LPS}_X[1] = \texttt{LPS}_Y[1] = i$. For every $i \leq h \leq n$, the substring $X[1..h]$ is a square if and only if $Y[1..h]$ is a square by Definition 8, and the statement holds trivially. For every $1 \leq j < i$, we prove the claim by contradiction. Without loss of generality, suppose that $X[1..j]$ is a square while $Y[1..j]$ is not. We separate the discussion into two cases, which together cover all possibilities.

To give a better illustration, we will use several figures as an aid. In these figures, we use *Color* to represent substrings in the figures. For instance, *Yellow* denotes the substring colored in yellow. Substrings sharing the same color represent identical strings, and additional labels are sometimes used to distinguish different occurrences of the same substring.

Case 1: $|X[1..j]| \leq \frac{1}{2}|X[1..i]|$. As shown in Figure 3(a), since $X[1..i]$ is a square, it follows that

$$X\left[\tfrac{i}{2}+1..\tfrac{i}{2}+j\right] = X[1..j].$$

Hence, the substring $X[\frac{i}{2}+1..\frac{i}{2}+j]$ is also a square.By the induction hypothesis, $Y[\frac{i}{2}+1..\frac{i}{2}+j]$ is a square as well. Moreover, since $Y[1..i]$ is a square, it follows that $Y[1..j]$ is a square, contradicting our assumption.

Case 2: $\frac{1}{2}|X[1..i]| < |X[1..j]|$. As shown in Figure 3(b), we have the following three squares:

Square 1: $X[1..\frac{j}{2}] = X[\frac{j}{2}+1..j]$.
Square 2: $X[1..\frac{i}{2}] = X[\frac{i}{2}+1..i]$.
Square 3: $Y[1..\frac{i}{2}] = Y[\frac{i}{2}+1..i]$.

Note that identical substrings are colored in the same color. To distinguish different occurrences of the same-colored substrings, we will use subscripts, such as Red_1, Red_2, and so on, for different substrings in red.

We first focus on the substring $X[\frac{j}{2}+1..\frac{i}{2}]$ (denoted by Red_1). By Square 1, Red_2 occurs as a prefix of X. Applying Square 2 and Square 1 repeatedly, we obtain Red_3 (by Square 2) and Red_4 (by Square 1). We continue this process until we obtain a substring whose ending position is greater than or equal to j (denoted by Red_5 in the example).

We divide Red_5 at position j into two substrings, denoted by U and V, and replace all occurrences of *Red* accordingly, as shown in Figure 3(b). We observe that there are multiple squares composed of U and V. These squares are marked by brackets of different colors in Figure 3(c). By the induction hypothesis, the corresponding substrings of Y are also squares and can be decomposed into P and W.

Finally, as shown in Figure 3(d), a sequence of P and W appears as a prefix of Y by Square 3. As indicated by the pink brackets, we have $Y[1..\frac{j}{2}] = Y[\frac{j}{2}+1..j]$, which implies that $Y[1..j]$ is a square, contradicting our assumption.

The case where V is empty, that is, when Red_5 ends exactly at position j, can be proved analogously and is therefore omitted.

In summary, both cases lead to a contradiction, and the proof completes. ◀

Observe that the LPS array of $S[i+1..i+\ell]$ coincides with the last ℓ entries of the LPS array of $S[i..i+\ell]$. This observation motivates the use of a two-dimensional table, called the *LPS table* ($\Lambda[1..n][1..n]$), to store the LPS arrays of all substrings of S.

▶ **Definition 10** (LPS table). *Let S be a string of length n. The two-dimensional table $\Lambda[1..n][1..n]$ is defined such that $\Lambda[\ell][i]$ represents the length of the* longest prefix square *of the substring $S[i..i+\ell-1]$.*

An example is illustrated in Figure 2(b). In particular, to reconstruct the LPS array of the substring $S[i..i+\ell-1]$, we consider the following diagonal of the LPS table:

$$\langle \Lambda[\ell][i], \Lambda[\ell-1][i+1], \ldots, \Lambda[1][i+\ell-1] \rangle.$$

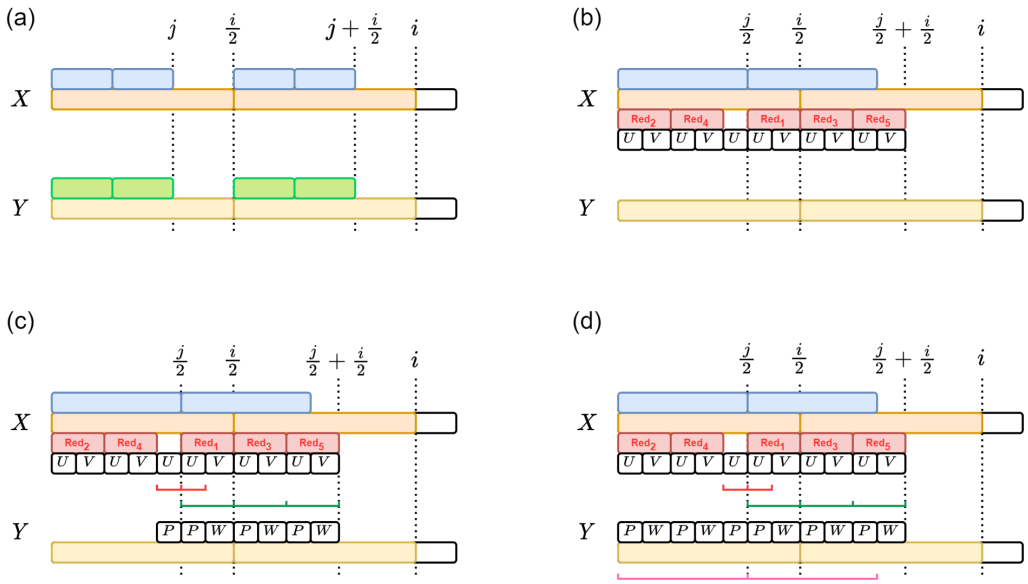

Figure 3 Examples illustrating the proof of Theorem 9: (a) corresponds to Case 1, and (b)–(d) correspond to different steps in Case 2.

The LPS table of S requires $n^2 \lg n$ bits of space to store and can be constructed in $O(n^2)$ time.[1] We defer the construction of the LPS table to Section 4.2.

Since the LPS table of S contains the LPS arrays of all substrings of S, we can answer $Loc_{sqr}(S, P)$ by locating $\texttt{LPS}_P$ among those represented in the LPS table of S. To support this operation efficiently, we construct an additional data structure: a trie built over all diagonals of the LPS table of S. Equivalently, this trie stores the LPS arrays of all prefixes of S.

▶ **Definition 11** (Diagonal Trie). *Let S be a string of length n, and let $\Lambda[1..n][1..n]$ denote the LPS table of S. For each position $i \in [1..n]$, consider the diagonal of Λ defined by*

$$\langle \Lambda[1][i], \Lambda[2][i-1], \ldots, \Lambda[i][1] \rangle.$$

The diagonal trie *of S is the compressed trie obtained by inserting all such diagonals, each terminated by a terminal symbol $\$$, one for each i.*

Each edge stores the corresponding start and end positions in $\Lambda[1..n][1..n]$. Each internal node stores pointers to the leftmost and rightmost leaves of its subtree. Each leaf stores the associated position i and a pointer to the next leaf in left-to-right order. The outgoing edges of each node are stored in a deterministic static linear-space dictionary [16], which supports worst-case $O(1)$ lookup time.

Note that the edges of the diagonal trie only store references to entries of the LPS table; thus, access to the LPS table must be available. The following lemma summarizes the space usage of the diagonal trie and its construction time.

▶ **Lemma 12.** *The diagonal trie of S requires $O(n \lg n)$ bits of space, in addition to the space required for the LPS table, and can be constructed in $O(n^2)$ time.*

[1] In practice, the space can be reduced by a factor of two by storing only rows corresponding to even lengths, since $\Lambda[\ell][i] = \Lambda[\ell - 1][i]$ for all odd ℓ.

Proof. The diagonal trie is a compressed trie containing n leaves, one for each diagonal of the LPS table (equivalently, one for each position of S). In a compressed trie, every internal node except the root has degree at least two. Hence, the number of internal nodes is at most $n-1$, and the total number of nodes and edges is $O(n)$.

Each node and edge stores indices or pointers of size $O(\lg n)$ bits. Since the total number of nodes and edges is $O(n)$, the trie itself occupies $O(n \lg n)$ bits. Moreover, the deterministic static dictionaries for the outgoing edges also require $O(n \lg n)$ bits in total. Therefore, the diagonal trie requires $O(n \lg n)$ bits of space.

To construct the trie, we insert all n diagonals of the LPS table. The total length of all diagonals is $\sum_{i=1}^{n} i = O(n^2)$, and thus the total insertion time is $O(n^2)$. Since each node is augmented with a deterministic static dictionary, and the total number of entries across all these dictionaries is $O(n)$, constructing all deterministic static dictionaries requires $O(n^{1+\epsilon})$ time [16] for any constant $\epsilon > 0$. We will choose $\epsilon < 1$. Since $O(n^{1+\epsilon}) = o(n^2)$, this does not affect the overall time bound. Therefore, the construction time is $O(n^2)$. ◀

Up to this point, the LPS index of S consists of the following two components:

1. the LPS table of S;
2. the diagonal trie of the LPS table.

To answer $Loc_{sqr}(S, P)$, we first compute $\texttt{LPS}_P$, and then traverse the diagonal trie of S by scanning $\texttt{LPS}_P$ from right to left. If the traversal succeeds and terminates at a node (which may be an implicit node located in the interior of an edge), then for each leaf in the subtree rooted at this node, we insert the position $i - |P| + 1$ into $Loc_{sqr}(S, P)$, where i is the position stored at the leaf. Otherwise, we return the empty set.

We summarize the query time complexity of the above procedure as follows.

▶ **Theorem 13.** *Given a string S of length n together with its LPS table and diagonal trie, and a query pattern P of length m, the set $Loc_{sqr}(S, P)$ can be computed in $O(m \lg m + occ)$ time, where occ denotes the number of substrings of S that square match with P. If the array $\texttt{LPS}_P$ is given, the time complexity can be reduced to $O(m + occ)$.*

Proof. Each node of the diagonal trie stores its outgoing edges in a deterministic static dictionary supporting worst-case $O(1)$ query time [16]. Thus, retrieving the next edge during traversal takes $O(1)$ time. Querying the pattern P traverses at most m nodes, resulting in a total time of $O(m)$. Traversing all leaves in the corresponding subtree takes $O(occ)$ time. If the array $\texttt{LPS}_P$ is not given, it can be computed in $O(m \lg m)$ time using Algorithm 1. Therefore, the overall time complexity is $O(m \lg m + occ)$. ◀

An example is illustrated in Figure 2(c). The blue labels v_i denote the node indices. Given $P = \texttt{cdd}$, we have $\texttt{LPS}_P = \langle 0, 2, 0 \rangle$. We start from the root (v_1), follow the edges to traverse v_2, and eventually reach v_4. From here on, we know that each leaf in the subtree rooted at v_4 witnesses a substring matching with P. Each leaf stores the ending position of this substring, so subtracting $|P| - 1$ from this ending position gives the starting position of the matching substring. By doing this for each leaf in this example, we form the set $Loc_{sqr}(S, P) = \{1, 4\}$.

4 Compression and Construction of LPS Index

The LPS index of S consists of the LPS table and the diagonal trie, where the former requires $O(n^2 \lg n)$ bits and the latter requires $O(n \lg n)$ bits. In this section, we first present a compressed representation of the LPS table using $O(n \lg^2 n)$ bits, and then describe how to construct the LPS index of S based on this compressed representation.

4.1 LPS Table Compression

We start from the well-known Three Squares Lemma (Lemma 2), which implies that there are only $O(\lg n)$ primitive squares in any fixed column of the LPS table of S. We exploit this property by storing only the corresponding $O(\lg n)$ entries, instead of all n entries.

We next introduce the following lemma to capture the relationship among these $O(\lg n)$ primitive squares.

▶ **Lemma 14.** *Let X and Y be two primitive strings. Suppose that X^{2k} is a non-primitive square and Y^{2t} is a square, both of which are prefixes of a string S, where $k \geq 2$ and $t \geq 1$. Then either $|Y| \leq |X|$ or $|Y| > |X^k|$ holds.*

Proof. Assume that $|X| < |Y| \leq |X^k|$. Since Y^2 has two periods $|X|$ and $|Y|$, and $|Y^2| \geq |X| + |Y|$, it follows from Lemma 1 that Y^2 has period $\gcd(|X|, |Y|)$. Hence, Y also has period $\gcd(|X|, |Y|)$, which implies that Y is not primitive, leading to a contradiction. ◀

Let $R_1, R_2, R_3, \ldots, R_j, R_{j+1}, \ldots$ be the roots of the $O(\lg n)$ primitive-square prefixes of $S[i..n]$, sorted by increasing length. Lemma 14 implies that any prefix square R_j^{2k} must be shorter than R_{j+1}^2. Therefore, for each primitive square, we only store the following information, which suffices to compute the LPS length of any entry in the column.

1. $|R_j|$, the root length of the j-th smallest primitive-square prefix of $S[i..n]$;
2. k_j, the maximum integer such that $R_j^{2k_j}$ is a prefix of $S[i..n]$.

For each column of the LPS table, we store at most $O(\lg n)$ such doublets in a contiguous array X by Lemma 2, and maintain the corresponding lengths $|R^2|$ in a dynamic fusion tree that supports $O(1)$-time rank queries using $O(\lg^2 n)$ bits of space [19].[2]

To retrieve the value of $\Lambda[q][i]$, we first compute the number of elements in the column $\Lambda[\cdot][i]$ that are less than or equal to q using a rank query. Suppose that the query returns p. We then retrieve the doublet $(|R_p|, k_p)$ stored in $X[p]$. Finally, we compute $\Lambda[q][i]$ as follows:

$$\Lambda[q][i] = \min\left(k_p, \left\lfloor \frac{q}{2|R_p|} \right\rfloor\right) \cdot 2|R_p|.$$

Since rank queries and access to the array X both take $O(1)$ time, it follows that $\Lambda[q][i]$ can be computed in $O(1)$ time. Therefore, the compressed LPS table preserves the query efficiency stated in Theorem 13.

4.2 LPS Index Construction

In this section, we first describe how to construct a compressed representation of the LPS table, where each column is represented using two auxiliary data structures: a dynamic fusion tree and an array X storing primitive-square doublets.

For a fixed column $\Lambda[\cdot][i]$ of the LPS table, we initialize the array X by setting

$$X[1] = (|R_1|, k_1) = (0, 0).$$

We then compute the failure function $f[1..n-i+1]$ of the string $S[i..n]$ using the KMP algorithm [12], where the value $f[t]$ is defined to be the length of the longest proper prefix of $S[i..i+t-1]$ that is also a suffix of $S[i..i+t-1]$. For $1 \leq \ell \leq n-i+1$, we compute the

[2] Since the set contains $O(\lg n)$ elements, each query takes $O(\lg \lg n / \lg \omega) = O(1)$ time in the word-RAM model with word size $w = \Theta(\lg n)$.

LPS of $S[i..i+\ell-1]$ using f as follows. Let $(|R_j|, k_j)$ be the last doublet stored in X. We first determine whether $S[i..i+\ell-1]$ is a square by checking whether $2(\ell - f[\ell])$ divides ℓ. If this condition does not hold, we proceed to the next value of ℓ. Otherwise, we check whether the root length $\ell - f[\ell]$ is equal to $|R_j|$. If so, we increment k_j by 1:

$$X[j] = (|R_j|, k_j + 1).$$

Otherwise, we append a new doublet:

$$X[j+1] = (|R_{j+1}|, k_{j+1}) = (\ell - f[\ell], 1).$$

Finally, we construct a dynamic fusion tree on the set of values $2|R_j| : (|R_j|, k_j) \in X$, thereby completing the construction of this column. After constructing the compressed LPS table, we build the diagonal trie as described in Lemma 12, which completes the construction of the LPS index.

We summarize the construction of the LPS index below.

▶ **Theorem 15.** *Given a string S of length n, the LPS index of S can be constructed in $O(n^2)$ time and occupies $O(n \lg^2 n)$ bits of space.*

Proof. For a fixed column $\Lambda[\cdot][i]$ of the LPS table, the associated information can be computed in $O(n)$ time by applying the KMP algorithm [12] to the suffix $S[i..n]$. The dynamic fusion tree corresponding to $\Lambda[\cdot][i]$ can also be constructed in $O(\lg n)$ time [19].[3] Since there are n columns in total, the entire LPS table can be constructed in $O(n^2)$ time. By Lemma 12, the diagonal trie can also be constructed in $O(n^2)$ time. Therefore, the total construction time of the LPS index is $O(n^2)$.

For a fixed column of the LPS table, the array X storing the primitive-square doublets requires $O(\lg^2 n)$ bits of space, and the dynamic fusion tree requires $O(\lg^2 n)$ bits of space. Therefore, summing over all n columns, the compressed LPS table requires $O(n \lg^2 n)$ bits of space. The diagonal trie requires $O(n \lg n)$ bits of space by Lemma 12. This completes the proof. ◀

5 Conclusion

In this paper, we introduced a new type of structural matching problem, called *Square Pattern Matching*. We propose the LPS index that requires $O(n \lg^2 n)$ bits of space and finds all *occ* occurrences matching with a pattern of length m in $O(m \lg m + occ)$ time. This running time can be reduced to $O(m + occ)$ time if the LPS array of the query pattern is given. This index is applicable to domains in which similarity in repetition structure is of interest, such as bioinformatics and music structure analysis.

Future work includes improving the time complexity of LPS array construction and exploring further compression of the LPS index.

References

1 Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003. doi:10.1016/S0890-5401(02)00035-4.

2 Brenda S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996. doi:10.1006/JCSS.1996.0003.

[3] A naïve approach inserts the $O(\lg n)$ elements one by one, each in $O(\lg \lg n / \lg w) = O(1)$ time (as $w = \Theta(\lg n)$).

3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

4 Maxime Crochemore and Wojciech Rytter. Squares, Cubes, and Time-Space Efficient String Searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

5 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

6 Jonas Ellert and Johannes Fischer. Linear Time Runs Over General Ordered Alphabets. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, Glasgow, Scotland (Virtual Conference), July 12-16, 2021*, pages 63:1–63:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.63.

7 Michael Fredman and Dan Willard. Blasting through the information theoretic barrier with fusion trees. In *ACM Symposium on Theory of Computing*, pages 1–7, 1990. doi:10.1145/100216.100217.

8 Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny M. Shur, and Tomasz Walen. String periods in the order-preserving model. *Inf. Comput.*, 270, 2020. doi:10.1016/J.IC.2019.104463.

9 Diptarama Hendrian. Generalized Dictionary Matching Under Substring Consistent Equivalence Relations. In *WALCOM: Algorithms and Computation (WALCOM 2020)*, volume 12049 of *Lecture Notes in Computer Science*, pages 120–132. Springer, 2020. doi:10.1007/978-3-030-39881-1_11.

10 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.

11 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/J.TCS.2013.10.006.

12 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

13 Roman M. Kolpakov and Gregory Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999, New York, NY, USA, October 17-18, 1999*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

14 Xingyu Liao, Wufei Zhu, Juexiao Zhou, Haoyang Li, Xiaopeng Xu, Bin Zhang, and Xin Gao. Repetitive DNA sequence detection and its role in the human genome. *Communications biology*, 6(1):954, 2023. doi:10.1038/s42003-023-05322-y.

15 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. doi:10.1016/J.TCS.2016.02.017.

16 Peter Bro Miltersen. Error Correcting Codes, Perfect Hashing Circuits, and Deterministic Dynamic Dictionaries. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 556–563. SIAM, 1998. URL: http://dl.acm.org/citation.cfm?id=314613.314845.

17 Shinya Nagashita and Tomohiro I. PalFM-Index: FM-Index for Palindrome Pattern Matching. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *LIPIcs*, pages 23:1–23:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CPM.2023.23.

18 Sung Gwan Park, Amihood Amir, Gad M. Landau, and Kunsoo Park. Cartesian Tree Matching and Indexing. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, volume 128 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.16.

19 Mihai Pătraşcu and Mikkel Thorup. Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

20 Jouni Paulus and Anssi Klapuri. Music structure analysis by finding repeated parts. In *Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia*, pages 59–68. Association for Computing Machinery, 2006. doi:10.1145/1178723.1178733.

21 Jouni Paulus, Meinard Müller, and Anssi Klapuri. State of the Art Report: Audio-Based Music Structure Analysis. In *International Society for Music Information Retrieval Conference (ISMIR 2010)*, pages 625–636. ISMIR, 2010. URL: http://ismir2010.ismir.net/proceedings/ismir2010-107.pdf.

22 Siwoo Song, Geonmo Gu, Cheol Ryu, Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast algorithms for single and multiple pattern Cartesian tree matching. *Theor. Comput. Sci.*, 849:47–63, 2021. doi:10.1016/J.TCS.2020.10.009.