String Indexing with Compressed Patterns

Philip Bille (ORCID)
Technical University of Denmark, DTU Compute, Denmark
phbi@dtu.dk

Inge Li Gørtz (ORCID)
Technical University of Denmark, DTU Compute, Denmark
inge@dtu.dk

Teresa Anna Steiner (ORCID)
Technical University of Denmark, DTU Compute, Denmark
terst@dtu.dk

---- **Abstract** ----

Given a string S of length n, the classic string indexing problem is to preprocess S into a compact data structure that supports efficient subsequent pattern queries. In this paper we consider the basic variant where the pattern is given in compressed form and the goal is to achieve query time that is fast in terms of the compressed size of the pattern. This captures the common client-server scenario, where a client submits a query and communicates it in compressed form to a server. Instead of the server decompressing the query before processing it, we consider how to efficiently process the compressed query directly. Our main result is a novel linear space data structure that achieves near-optimal query time for patterns compressed with the classic Lempel-Ziv 1977 (LZ77) compression scheme. Along the way we develop several data structural techniques of independent interest, including a novel data structure that compactly encodes all LZ77 compressed suffixes of a string in linear space and a general decomposition of tries that reduces the search time from logarithmic in the size of the trie to logarithmic in the length of the pattern.

1 Introduction

The string indexing problem is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we introduce a basic variant of string indexing, called the *string indexing with compressed pattern problem*, where the pattern P is given in compressed form and we want to answer the query without decompressing P. The goal is to obtain a compact structure while achieving fast query times in terms of the compressed size of P.

The string indexing with compressed pattern problem captures the following common client-server scenario: a client submits a query and sends it to a server which processes the query. To minimize communication time and bandwidth the query is sent in compressed form. Naively, the server will then have to decompress the query and then process it. With an efficient solution to the string indexing with compressed pattern problem we can eliminate the overhead decompression and speed up queries by exploiting repetitions in pattern strings.

To the best of our knowledge, no non-trivial solution to the string indexing with compressed pattern problem are known. In contrast, the opposite problem, where the indexed string S is compressed and the pattern P is uncompressed, is well-studied [18, 17, 6, 21, 15, 7, 8, 13, 14, 9, 26, 22, 5, 20, 23, 11, 2, 3] (see also the surveys [26, 24, 25, 12]).

We focus on the classic Lempel-Ziv 1977 (LZ77) [29] compression scheme. Note that since the size of an LZ77 compressed string is a lower bound for many other compression schemes (such as all grammar-based compression schemes) our results can be adapted to such compression schemes by recompressing the pattern string. To state the bounds, let n be the length of S, m be the length of P, and z be the LZ77 compressed length of P. Naively, we can solve the string indexing with compressed pattern problem by using a suffix tree of S as our data structure and answering queries by first decompressing them and then traversing the suffix tree with the uncompressed pattern. This leads to a solution with $O(n)$ space and $O(m + \text{occ})$ query time. At the other extreme, we can store a trie of all the LZ77 compressed suffixes of S together with a simple tabulation, leading to a solution with $O(n^3)$ space and $O(z + \text{occ})$ query time (see discussion in Section 3).

We present the first non-trivial solution to the string indexing with compressed pattern problem achieving the following bound:

▶ **Theorem 1.** *We can solve the string indexing with compressed pattern problem for* LZ77-*compressed patterns in $O(n)$ space and $O(z + \log m + \text{occ})$ time, where n is the length of the indexing string, m is the length of the pattern, and z is the number of phrases in the* LZ77 *compressed pattern.*

Since any solution must use at least $\Omega(z + \text{occ})$ time to read the input and report the occurrences, the time bound in Theorem 1 is optimal within an additive $O(\log m)$ term. In the common case when $z = O(\log m)$ or if we consider LZ77 without *self-references* the time bound is optimal. For simplicity, we focus on reporting queries, but the result is straightforward to extend to also support *existential queries* (decide if the pattern occurs in S) and *counting queries* (count the number of occurrences of the pattern in S) in $O(z + \log m)$ time and the same space.

To achieve Theorem 1 we develop several data structural techniques of independent interest. These include a compact data structure that encodes all LZ77 compressed suffixes of a string in linear space and a general decomposition of tries that reduces the search time from logarithmic in the size of the trie to logarithmic in the length of the pattern.

The paper is organized as follows. In Section 2 we recall basic string data structures and LZ77 compression. In Section 3 we present a simple $O(n^2)$ space and $O(z + \log n + \text{occ})$ time data structure that forms the basis of our solutions in the following sections. In Section 4 we show how to achieve linear space with the same time complexity. Finally, in Section 5 we show how to improve the $\log n$ term to $\log m$.

2   Preliminaries

A string S of length n is a sequence $S[0] \cdots S[n-1]$ of n characters drawn from an alphabet Σ. The string $S[i] \cdots S[j-1]$ denoted $S[i, j]$ is called a *substring* of S. The substrings $S[0, j]$ and $S[i, n]$ are called the jth *prefix* and i^{th} *suffix* of S, respectively. We will sometimes use S_i to denote the ith suffix of S.

Longest Common Prefix

For two strings S and S', the *longest common prefix of S and S'*, denoted $\mathtt{lcp}(S, S')$, is the maximum $j \in \{0, \dots, \min(|S|, |S'|)\}$ such that $S[0, \ j] = S'[0, \ j]$.

Given a string S of length n, there is a data structure of size $O(n)$ that answers $\mathtt{lcp}$-queries for any two suffixes of S in constant time by storing a suffix tree combined with an efficient nearest common ancestor (NCA) data structure [16, 28].

Compact Trie

A compact trie for a set D of strings $S^1, \dots, S^l$ is a rooted labeled tree T_D, with the following properties: The label on each edge is a substring of one or more S^i. If the set of strings is prefix free, each root-to-leaf path represents a string in the set (obtained by concatenating the labels on the edges of the path), and for every string there is a leaf corresponding to that string. Common prefixes of two strings share the same path maximally, and all internal vertices have at least two children.

The compact trie has $O(l)$ nodes and edges and a total space complexity of $O\left(\sum_{i=1}^{l} |S^i|\right)$. The position in the trie that corresponds to the maximum longest common prefix of a pattern P of length m and any S^i can be found in $O(m)$ time. For a position p in the tree, which can be either a node or a position within the label of an edge, let $\mathrm{str}(p)$ denote the string obtained by concatenating the labels on the path from the root to p. The locus of a string P in T_D, denoted $\mathtt{locus}(P)$, is the deepest position p in the tree such that $\mathrm{str}(p)$ is a prefix of P. A compact trie on the suffixes of a string S is called the *suffix tree* of S and can be stored in linear space [28]. The *suffix array* stores the starting positions of the suffixes in the string in lexicographic order. If at every node in the suffix tree its children are stored in lexicographic order, the order of the suffix array corresponds to the order of the leaves in the suffix tree.

LZ77

Given an input string S of length n, the LZ77 parsing divides S into z substrings $f_1, f_2, \dots, f_z$, called phrases, in a greedy left-to-right order. The ith phrase f_i, starting at position p_i is either (a) the first occurrence of a character in S or (b) the longest substring that has at least one occurrence starting to the left of p_i. If there are more than one occurrence, we assume that the choice is made in a consistent way. To compress S, we can then replace each phrase f_i of type (b) with a pair (r_i, l_i) such that r_i is the distance from p_i to the start of the previous occurrence, and l_i is the length of the phrase. The occurrence of f_i at position $p_i - r_i$ is called the *source* of the phrase. (This is actually the LZ77-variant of Storer and Szymanski [27]; the original one [29] adds a character to each phrase so that it outputs triples instead of tuples.)

Every LZ77-compressed string is a string over the extended alphabet which consists of all possible LZ77 phrases. For any string T we denote this string by $\mathrm{LZ}(T)$.

3   A Simple Data Structure

In this section we will define a data structure that allows us to solve the string indexing with compressed pattern problem in $O(n^2)$ space and $O(z + \log n + \mathrm{occ})$ time, or $O(n^3)$ space and $O(z + \mathrm{occ})$ time. This data structure forms the basis of our solution.
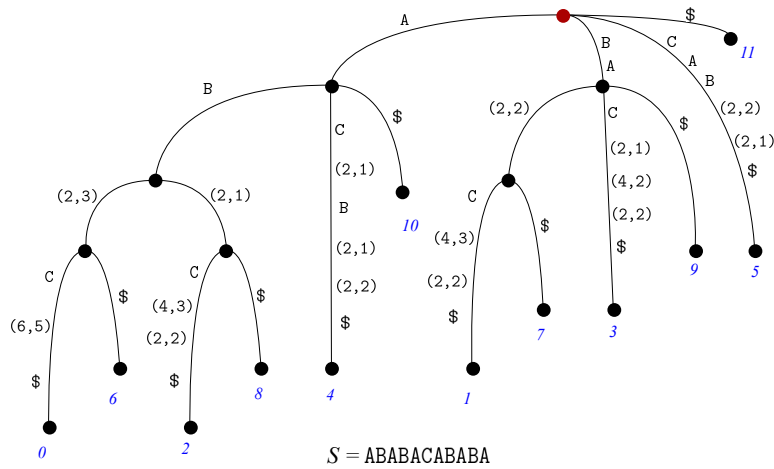
Figure 1 The phrase trie for the string `ABABACABABA$`. In this example, the leaves are sorted according to the lexicographic order of the originial suffixes. For instance the 6th suffix `ABABA$` has the LZ77 parse `A B (2,3) $`, and this string corresponds to the concatenation of labels on the path from the root to the second leaf.

The Phrase Trie

The *phrase trie of a string* S is defined as the compact trie over the set of strings $\{\mathrm{LZ}\,(S_i\$)\,, i = 0, \ldots, |S| - 1\} \cup \{\$\}$, that is, the LZ77 parses of all suffixes of S appended by a new symbol $\$$ which is lexicographically greater than any letter in the alphabet. For an example see Figure 1.

The phrase trie for a string S of length n has $n + 1$ leaves, one corresponding to every suffix of $S\$$. Similarly as in the suffix tree, every internal node defines a consecutive range within the suffix array. Since every node has at least 2 children the number of nodes and edges is $O(n)$.

LZ77 has the property that for two strings whose prefixes match up to some position ℓ the LZ77-compression of the two strings will be the same up to (not necessarily including) the phrase that contains position ℓ. As such, we can use the phrase trie to find the suffix S_i of S for which the LZ77-compression of the pattern P agrees with the LZ77-compression of S_i for as long as possible. Assuming they match for $k - 1$ phrases, the longest match of P in S ends within the kth phrase. If we additionally to the phrase trie keep a table for all possible phrases of a pattern that the longest match could end in, encoded as the triple (p, r, l) of starting position, distance to the start of the previous occurance, and length of the phrase, and which for each such phrase stores the solution to the query, we can solve the string indexing with compressed pattern problem in $O(n^3)$ space and $O(z + \mathrm{occ})$ time. Instead, we will store a linear space and constant time `lcp` data structure for S and show that given the first phrase where the suffix S_i and the string P mismatch, we can find the `lcp` of P and S_i by finding the `lcp` of two substrings of S.
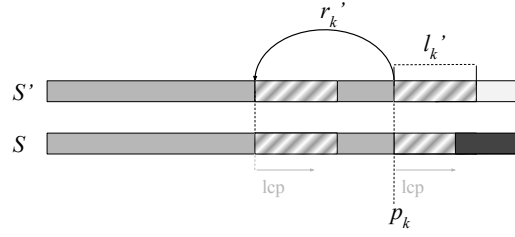
■ **Figure 2** The kth phrase in S' is copied from position $p_k - r'_k$, at which point S and S' are identical; the `lcp` value gives how far p_k and $p_k - r'_k$ match in S.

Longest Common Prefixes in LZ77-Compressed Strings

We will use an intuitive property about LZ77-compressed strings: assuming two strings match up until a certain phrase $k - 1$, we can reduce the task of finding the `lcp` of the two strings to the task of finding the longest common prefix between two suffixes of one of the strings. This property is summarized in the following lemma (see also Figure 2):

▶ **Lemma 2.** *Let* $S = f_1 \, f_2 \, \cdots \, f_z$ *and* $S' = f'_1 \, f'_2 \, \cdots \, f'_{z'}$ *be two strings parsed into LZ77 phrases, where* $f_1 = f'_1$, $f_2 = f'_2, \ldots, f_{k-1} = f'_{k-1}$. *Let* p_k *be the starting position of* f_k *and* f'_k. *If* f'_k *is a phrase represented by a pair* (r'_k, l'_k) *the following holds:*

$$\texttt{lcp}\,(S, S') \geq p_k + \min\left(\texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n])\,, l'_k\right). \tag{1}$$

Furthermore, if $f_k \neq f'_k$, *equality holds in (1).*

Proof. To prove (1), we will show by induction that for any

$$i \leq \min\left(\texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n])\,, l'_k\right),$$

we have that $S[p_k + i - 1] = S'[p_k + i - 1]$. For $i = 0$ this is true since S and S' are the same up until position $p_k - 1$. For the induction step assume it is true for all $i_0 < i$. We then have

$$S'[p_k + i - 1] = S'[p_k - r'_k + i - 1] \tag{2}$$
$$= S[p_k - r'_k + i - 1] \tag{3}$$
$$= S[p_k + i - 1], \tag{4}$$

where (2) follows from $i \leq l'_k$ and because $p_k - r'_k$ is the source of phrase f'_k, (3) follows from the induction hypothesis, and (4) follows from $i \leq \texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n])$.

To show equality in the case where $f_k \neq f'_k$, let $t = \min\left(\texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n])\,, l'_k\right)$. We will show that $S[p_k + t] \neq S'[p_k + t]$. There are two cases:

For $t = \texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n]) < l'_k$, note that $S[p_k - r'_k + t] \neq S[p_k + t]$. From (1) we know that $S'[p_k - r'_k + t] = S[p_k - r'_k + t]$, and therefore we have $S'[p_k + t] = S'[p_k - r'_k + t] = S[p_k - r'_k + t] \neq S[p_k + t]$.

For $l'_k \leq \texttt{lcp}\,(S\,[p_k, \ n]\,, S\,[p_k - r'_k, \ n])$, note that by (1), we know that S and S' have an `lcp` of length at least $p_k + t$. If $t \geq l_k$, then by the uniqueness of the greedy left-to-right parsing, the kth phrase of S and S' would be the same, contradicting our condition. Otherwise, we have $l_k > t = l'_k$. This together with (1) implies $S[p_k + i] = S[p_k + i - r_k] = S'[p_k + i - r_k]$ for every $i = 0, \ldots, t$, since $r_k \geq 1$. By the greedy parsing property and since $l'_k = t$ we know that $S'[p_k + t - r_k] \neq S'[p_k + t]$ and so $S[p_k + t] \neq S'[p_k + t]$.   ◀

3.1   The Data Structure

Additionally to storing the phrase trie of S, we store the suffix array of S, and for every node in the phrase trie, the range of the leaves below it in the suffix array. Finally, we store a linear space and constant time data structure for answering `lcp`-queries for suffixes of S.

3.2   Algorithm

We begin by matching $\text{LZ}(P)$ as far as possible in the phrase trie. Let $v = \texttt{locus}(\text{LZ}(P))$. Let k be the first phrase in $\text{LZ}(P)$ that does not match any of the next phrases in the trie. If v is a node set $w = v$, otherwise let w be the first node below v. We proceed as follows:

- If the kth phrase in P is a single letter, we return p_k as the length of the match and the interval of positions stored at w.
- If the kth phrase is represented by (r_k, l_k) then there are two cases:
    - If v is on an edge, let S_i be the suffix corresponding to any leaf below v. We return

    $$p_k + \min(\texttt{lcp}\,(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n])\,, l_k)$$

    as the length of the match and the interval of positions stored at w.
    - If v is on a node, we do a binary search for the longest match in the range in the suffix array below v. That is, for the suffix S_i corresponding to the middle leaf in the range below v, we compute $\texttt{lcp}(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n])$. If this is greater than l_k we stop the binary search. Otherwise, we check if the next position in suffix S_i is lexicographically smaller or bigger than the next position in P to see whether we go left or right in the binary search. That is, we compare $S_i[t] = S[i + t]$ with $P[t] = S_i[t - r_k] = S[i + t - r_k]$, and update our search accordingly. We also keep track of the longest match found so far. At the end of the search, we go to longest match, and check left and right in the suffix array to find all occurrences.

3.3   Correctness

The compact trie gives us the longest matching prefix of $\text{LZ}(P) = f_1 \dots f_{z_p}$ in the phrase trie. That is, we find all suffixes $S_i = f'_1 \cdots f'_{z_i}$ for $i = 0, \dots, n-1$ such that $f_1 = f'_1, \dots, f_{k-1} = f'_{k-1}$ and $f_k \neq f'_k$, and k is maximal. By the uniqueness of parsing, the longest prefix of P found in S is the prefix of at least one these suffixes.

Note that by the greedy parsing, the longest match of the kth phrase has to end before the next node in the trie. We argue the different cases:

If the kth phrase in P is a letter, it did not appear in P before. Thus, it never appeared in any of the suffixes we matched so far. Since the next phrase in the phrase trie is different, it is either a copied position, or a different letter. In any case, the next letter of any candidate suffix does not match the next letter in P.

If f_k is represented by (r_k, l_k) there are two subcases. If v is on an edge, recall that S_i is the suffix corresponding to any leaf below the current position v. By Lemma 2 and since $S_i[p] = S[p + i]$ for any p, we have that

$$\begin{aligned} \texttt{lcp}\,(S_i, P) &= p_k + \min(\texttt{lcp}\,(S_i\,[p_k,\ n]\,, S_i\,[p_k - r_k,\ n])\,, l_k) \\ &= p_k + \min(\texttt{lcp}\,(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n])\,, l_k). \end{aligned}$$

If v is on a node we have, by the same argument as before,

$$\texttt{lcp}\,(S_i, P) = p_k + \min(\texttt{lcp}\,(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n])\,, l_k),$$

for every suffix S_i. Further, because of the lexicographic order of the suffix array, we can binary search to find the leaf with the longest match, and by checking the adjacent positions in the suffix array we make sure to find all occurrences.

3.4   Analysis

The suffix array and the `lcp` data structure both use linear space in the size of S. For the phrase trie, we store the LZ77-compressed suffixes of S, which use $O(\sum_{i=1}^{n} z_i) = O(n^2)$ space, where z_i is the number of phrases used to compress suffix S_i.

For the time complexity, we use $O(k) = O(z)$ time for matching the phrases in the trie. In the worst case, that is, when the locus v is on a node, we need $O(\log(\#\texttt{leaves below } v)) = O(\log n)$ constant time `lcp` queries. In total, we have a time complexity of $O(z + \log n + \text{occ})$. In summary, we proved the following lemma.

▶ **Lemma 3.** *The phrase trie solves the string indexing with compressed pattern problem in $O(n^2)$ space and $O(z + \log n + \text{occ})$ time.*

4   Space Efficient Phrase Trie

In this section, we show how to achieve the same functionality as the phrase trie while using linear space. The main idea is to store only one phrase per edge, and use Lemma 2 to navigate along an edge. That is, we no longer store the entire LZ77-compressed suffixes of S.

4.1   The Data Structure

We store a compact form of the phrase trie, which is essentially a blind trie version of the phrase trie. We store the following: We keep the tree structure of the phrase trie, and at each node, we keep a hash table, using perfect hashing [10], where the keys are the first LZ77 phrase of each outgoing edge. For each edge we store as additional information the length of the (uncompressed) substring on that edge and an arbitrarily chosen leaf below it. For an example see Figure 3. As before, we additionally store the suffix array, the range within the suffix array for each node, and a linear-sized `lcp` data structure for S.
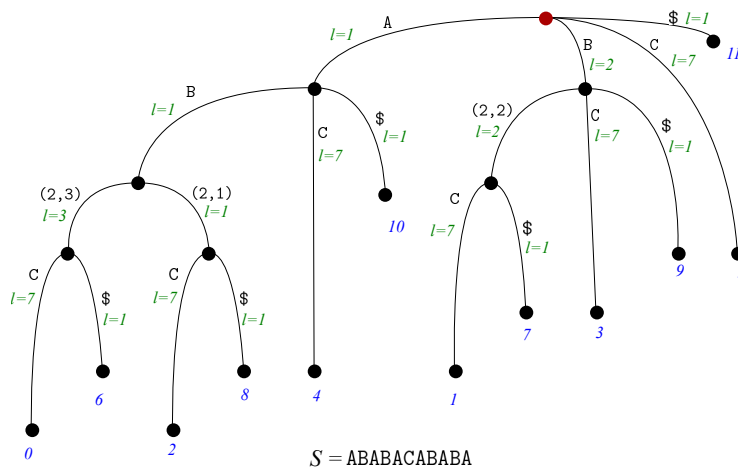
Figure 3 The phrase trie for the string `ABABACABABA` using linear space.

4.2   Algorithm

The algorithm proceeds as follows. We start the search at the root. Assume we have matched $k-1$ phrases of P and the current position in the trie is a node v. To match the next phrase we check if the kth phrase in P is in the hash table of v.

1. If it is not, we proceed exactly as in the previous section in the case where the locus is at a node.

2. If the kth phrase is present, let e be the corresponding edge and let i be the starting index of the leaf stored for e. Set $k = k + 1$. We do the following until we reach the end of edge e or get a mismatch. We differentiate between two cases.

   - The kth phrase is a single letter α:
     - If $\alpha = S[i + p_k]$, we set $k = k + 1$ and continue with the next phrase.
     - If $\alpha \neq S[i + p_k]$, we stop and return p_k as the length of the match.
   - The kth phrase is represented by (r_k, l_k):
     - If $\min(\texttt{lcp}(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n]), l_k)) \geq l_k$, we set $k = k+1$ and continue with the next phrase.
     - Otherwise, we return $p_k + \texttt{lcp}(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n])$ as the length of the match, with the interval of positions stored at the next node.

   If we reach the end of an edge, we go to the next node below and continue in the same way.

Correctness

The correctness follows from the previous section together with Lemma 2, since we always keep the invariant that when we process the kth phrase, we already matched the $k-1$ previous ones.

Analysis

The space complexity is linear since the compact phrase trie has $O(n)$ nodes and edges and stores constant information per node and edge, using perfect hashing.

    The time complexity is the same as in the previous section, since for matching full phrases, we use at most one constant time lookup in the hash table and one constant time $\texttt{lcp}$ query per phrase in P. As before, the worst case for matching the kth phrase is having to do a binary search, using $O(\log n)$ constant time $\texttt{lcp}$ queries. In summary, this gives the following lemma.

▶ **Lemma 4.** *We can solve the string indexing with compressed pattern problem in $O(n)$ space and $O(z + \log n + \mathrm{occ})$ time.*

5   Slice Tree Solution

In this section, we show how to reduce the $O(\log n)$ time overhead to $O(\log m)$. Recall that the additional $O(\log n)$ time originates from the binary search in the case where after matching $k-1$ phrases we arrive at a node, and the kth phrase does not match any of the outgoing edges. In any other case, the solution from the previous section gives $O(z + \mathrm{occ})$ time complexity. We use the solution from the previous section as a basis and show how to speed up the last step of matching the kth phrase. For our solution, we use Karp-Rabin fingerprints and the ART tree decomposition, which we define next.

Karp-Rabin Fingerprints

For a prime p and an $x \leq p$, the *Karp-Rabin fingerprint* [19] of a substring $S\,[i,\ j]$ is defined as

$$\phi_{p,x}(S\,[i,\ j]) = \sum_{k=i}^{j-1} S[k]x^{k-i} \mod p.$$

Clearly, we have that if $S\,[i,\ j] = S'\,[i',\ j']$, then $\phi_{p,x}(S\,[i,\ j]) = \phi_{p,x}(S'\,[i',\ j'])$. Furthermore, the Karp-Rabin fingerprint has the property that for any three strings x, y and z where $z = xy$, given the fingerprint of any two of those strings, the third one can be computed in constant time. It follows that given the fingerprints of all suffixes of a string S, the fingerprint of any substring of S can be computed in constant time.

We assume that p and x are chosen in such a way that $\phi_{p,x}$ is *collision-free* on substrings of S, that is, two distinct substrings of S have different fingerprints. For details on how to construct $\phi_{p,x}$ see for example [4]. We will from now on use the notation $\phi = \phi_{p,x}$.

ART decomposition

The ART decomposition of a tree by Alstrup et al. [1] partitions a tree into a *top tree* and several *bottom trees*. Every vertex v of minimal depth with no more than χ leaves below it is the root of a bottom tree which consists of v and all its descendants. The top tree consists of all vertices that are not in any bottom tree. The following lemma gives a key property of ART trees:

▶ **Lemma 5** (Alstrup et al.[1]). *The ART decomposition with parameter χ for a rooted tree T with n leaves produces a top tree with at most $\frac{n}{\chi+1}$ leaves.*

5.1 The Slice Tree Decomposition

The overall idea is to construct a two level decomposition of the suffix tree. First, we will divide the tree into smaller trees, the *slice trees*, where the heights are powers of two and increase with the depth in the tree. Each of those slice trees is decomposed using an ART decomposition. Together with Karp-Rabin fingerprints stored at the roots of each slice tree, this will allow us to efficiently carry out an approximate search for the longest match, so we can then use the slice trees to find the exact position and length.

In more detail, we store the space efficient phrase trie from the previous section for matching full phrases of the pattern. Additionally, we store the Karp-Rabin fingerprints for each suffix of S, as well as the following *slice tree decomposition* of the suffix tree of S:

- We store the suffix tree together with extra nodes at any position in the suffix tree that corresponds to a string depth that is a power of two. For each node we store the range in the suffix array of the leaves below.
- For each level of string depth 2^i, where $i = 0, \ldots, \lfloor \log n \rfloor$, we store a static hash table with Karp-Rabin fingerprints of the substring in S from the root to every node of string depth 2^i. As in section 4, we use perfect hashing for all hash tables in this solution.
- For each node v at string depth 2^i we define a *slice tree* of order i. The slice tree is the subtree rooted at v, cut off at string depth 2^i, such that the string height of the slice tree is (at most) 2^i.
- We compute an ART decomposition of each slice tree of order i with the parameter χ set to $\chi = 2^i$. For each $1 \leq d < 2^i$, we store a hash table with fingerprints corresponding to the substrings of length d starting at the root of the slice tree and ending in the top tree.

Additionally, for every edge connecting a top tree node to a bottom tree root save the corresponding first letter in the suffix tree. For every leaf in the bottom tree we store the starting position of a leaf below it in the suffix tree.

5.2   Algorithm

To match P, we first match the full phrases in the phrase trie until we find the first phrase f_k which does not match any of the next phrases in the trie. If f_k is just a letter, as before, we are done. Otherwise f_k is represented by (r_k, l_k). Now:

- We find the fingerprint $\phi(P[0, p_k]) = \phi(S[i_0, i_0 + p_k])$, where i_0 is a leaf below the current position in the phrase trie. Note that since $S[i_0, i_0 + p_k]$ is a substring of S and we stored the fingerprints of all suffixes of S we can find its fingerprint in constant time via the fingerprints of the suffixes S_{i_0} and $S_{i_0 + p_k}$.
- In order to find the slice tree where the match ends, we do a linear search for the deepest matching fingerprint in the hash tables at the power of 2 levels in the following way:
    - For $j \in \left\{ 2^{\lceil \log p_k \rceil} - p_k, 2^{\lceil \log p_k \rceil + 1} - p_k, \ldots, 2^{\lfloor \log n \rfloor} - p_k \right\}$ and while $j < l_k$, find the fingerprint of $f_k[0, j] = S[i_0 + p_k - r_k, i_0 + p_k - r_k + j]$ and look for $\phi(P[0, p_k]) + x^{p_k} \phi(f_k[0, j]) \mod p$ in the hash table of depth $p_k + j$. If $\phi(P[0, p_k]) + x^{p_k} \phi(f_k[0, j]) \mod p = \phi(S_i)$ for some i, we check if $\phi(S[i, i + p_k]) = \phi(P[0, p_k])$ to avoid false positives. We keep doing this until the first level where it is not present or the check fails.
    - For the last level where there is a match, we find the corresponding node the slice tree rooted at that node. Note that this slice tree can be of order at most $\log m$.
- Similarly as the linear search above, we now do an exponential search for fingerprints on the levels in the top tree of the slice tree. For the lowest level in which there is a match in the top tree, find the corresponding position v. If this is an internal node without any off-hanging bottom trees or on an edge in the top tree then $\texttt{locus}(P) = v$. Once we have found $\texttt{locus}(P)$ we can easily find and return the occurrences as before. Otherwise, we check if the next letter in P matches any of the off-hanging bottom trees. Again, we can find this letter in constant time by looking up its source in S. If it matches, we do a binary search for the longest match with the leaves of the bottom tree, which proceeds exactly as in the phrase trie solution, but restricted to the representative leaves stored for each bottom tree leaf. For each bottom tree leaf that has a longest match with P report all suffix tree leaves below it.

5.3   Correctness

The correctness of matching the first $k - 1$ phrases follows from the previous section. Given that k is the first phrase that does not match any of the next phrases in the suffixes, we argue for the linear search in the power of two levels in the suffix tree. We know that the Karp-Rabin fingerprints have no false negatives, so if $P[0, p_k + j] = S[i, i + p_k + j]$ for some i, then $\phi(P[0, p_k]) + x^{p_k} \phi(f_k[0, j]) \mod p$ will be present in the hash table of level $p_k + j$. Further, we chose ϕ such that it has no false positives on substrings of S, so by checking $\phi(P[0, p_k]) = \phi(S[i, i + p_k])$ separately, we make sure that $P[0, p_k + j]$ and $S[i, i + p_k + j]$ are actually identical. Together, this means that by finding the biggest j such that $p_k + j$ is a power of two and both conditions are fulfilled, we will find the slice tree that contains the end of the longest match.

Next, we argue for the detailed search within the slice tree. The argument for the exponential search is the same as for the linear search. When we end the exponential search, we found the position in the top tree of maximum depth that corresponds to a substring

of S matching a prefix of P. So the longest match either ends there or in a bottom tree that is connected to this position. If there is more than one such bottom tree, the first letter on each edge will uniquely identify the bottom tree that contains the leaf or leaves with the longest match. If the longest match ends in a bottom tree, it is enough to do the binary search with any representative leaf in the suffix tree per leaf in the bottom tree, since for any such leaf the prefix of a given length that ends in the bottom tree is the same.

5.4   Analysis

We use linear space for the phrase trie representation of the previous section and the fingerprints of the suffixes of S. Additionally, we use $O(n \log n)$ space for the extra nodes and hash tables at the power of two levels.

For each slice tree T of order i denote $|T|$ the number of nodes in the slice tree and let $h = 2^i$ be the maximal height of the slice tree. By Lemma 5, the top tree has at most $|T|/h$ leaves. By the definition of the slice tree, each root-to-leaf path has at most h positions. As such, the hash tables for the top tree take up $O(|T|)$ space. Furthermore we use constant space per leaf in the bottom tree. Each bottom tree leaf is a node in the suffix tree or an extra node, and each such node is a leaf in at most one bottom tree. So the total space for all slice trees is $\sum_{T \text{ is slice tree}} O\left(|T|\right) = O(\#\texttt{nodes in suffix tree} + \texttt{extra nodes}) = O(n \log n)$.

For the time complexity, as before, we use $O(z)$ for matching in the phrase trie. Since we stored the fingerprints of all suffixes of S, the fingerprint of any substring of S can be found in constant time.

For the linear search of fingerprints in the suffix tree, note that the last phrase of P is at most m long. This means we stop the search after checking at most $\log m$ power of 2 levels, and a check can be done in constant time.

After the linear search we end up in a slice tree of order at most $\log m$, which means $h \le m$. It follows that the exponential search in the top tree uses time at most $O(\log h) = O(\log m)$. Further, by the definition of the ART decomposition, every bottom tree has no more than $h \le m$ leaves, and as such the binary search in the bottom tree uses no more than $O(\log m)$ operations.

In total, this gives us a time complexity of $O(z + \log m + \text{occ})$. We arrive at the following result:

▶ **Lemma 6.** *The slice tree solution solves the string indexing with compressed pattern problem in $O(n \log n)$ space and $O(z + \log m + \text{occ})$ time.*

6   Saving Space

For the solution above, we constructed $O(n \log n)$ slice trees. By the way we defined them, note that any internal node in a slice tree has to be an original node from the suffix tree. Since there are only $O(n)$ such nodes, we conclude that many of the slice trees consist of a single edge. We will show that by removing those, we can define a linear space solution that gives the same time complexity as in Lemma 6.

6.1   The Data Structure

We start with the slice tree solution. Call every edge that contains two or more extra nodes a *long edge*. For every long edge, delete every extra node except the first and last, which we call v_{first} and v_{last}. For every deleted node also delete the additional information stored for their slice trees, and their corresponding entries in the power of two hash tables. For each long edge, store at the hash table position of v_{first} additionally the information that it is on a long edge, how long that edge is, and a leaf below it.

6.2 Algorithm

The algorithm proceeds almost as before. The only change is that in the linear search of power of two levels, when we match with a node that is v_{first} of a long edge, jump directly to the last power of two level that is before the end of the edge. If the fingerprint is present, proceed normally, otherwise, the longest match ends on that edge and we do a single `lcp` query between the source of the phrase in S and the stored leaf to find its length.

6.3 Correctness

If we do not encounter any long edges, nothing changes. If a long edge is entirely contained in the match, we will first find v_{first} and then jump directly to the last power of two level on that edge, where we will find v_{last}, and then continue as before. If the longest match ends on a long edge, there are two cases:

1. The longest match ends before v_{first} or after v_{last}: this means that by doing the linear search we find the slice tree that the longest match ends in, thus everything follows as before.
2. The longest match ends between v_{first} and v_{last}: In this case, we will find a matching fingerprint at the level corresponding to v_{first} but no matching fingerprint at the level corresponding to v_{last}, which means we will use `lcp` to find the longest match with a leaf below v_{first}. Since the match ends on that edge, this gives us the correct length and position.

6.4 Analysis

For space complexity, note that we only keep original nodes from the suffix tree, plus at most two extra nodes per edge, so a linear number of nodes in total. Since the space used for the slice trees and power of two hash tables is linear in the number of nodes, the total space consumption is linear. The time complexity does not change. This concludes the proof of Theorem 1.

───── **References** ─────

1   Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–543, 1998.

2   Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4):23, 2014.

3   Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time–space trade-offs for lempel–Ziv compressed indexing. *Theoret. Comput. Sci.*, 713:66–77, 2018.

4   Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Proc. 26th CPM*, pages 65–76, 2015.

5   Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, pages 180–192, 2012.

6   Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st FOCS*, pages 390–398, 2000.

7   Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proc. 12th SODA*, pages 269–278, 2001.

8   Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

9   Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.

10   Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

11   Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, pages 731–742, 2014.

12   Travis Gagie and Simon J Puglisi. Searching and indexing genomic databases via kernelization. *Front. Bioeng. Biotechnol.*, 3:12, 2015.

13   Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

14   Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th SODA*, pages 636–645, 2004.

15   Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

16   Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

17   Juha Kärkkäinen and Erkki Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

18   Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd WSP*, pages 141–155, 1996.

19   Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev*, 31(2):249–260, 1987.

20   Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoret. Comp. Sci.*, 483:115–133, 2013.

21   Veli Mäkinen. Compact suffix array. In *Proc. 11th CPM*, pages 305–319, 2000.

22   Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Bio.*, 17(3):281–308, 2010.

23   Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.

24   Gonzalo Navarro. Indexing highly repetitive collections. In *Proc. 23rd IWOCA*, pages 274–279, 2012.

25   Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

26   Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007.

27   James A Storer and Thomas G Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

28   Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th FOCS*, pages 1–11, 1973.

29   Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, 1977.