String Indexing with Compressed Patterns

PHILIP BILLE, INGE LI GØRTZ, and TERESA ANNA STEINER, Technical University of Denmark, DTU Compute, Denmark

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 article, 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.

CCS Concepts: • **Theory of computation → Data compression**; **Pattern matching;**

Additional Key Words and Phrases: String indexing

ACM Reference format:
Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. 2023. String Indexing with Compressed Patterns. *ACM Trans. Algor.* 19, 4, Article 32 (September 2023), 19 pages.
https://doi.org/10.1145/3607141

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 article, 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 that processes the query. To

An extended abstract appeared at the *37th Symposium on Theoretical Aspects of Computer Science (STACS2020)* [6].

Supported by the Danish Research Council (DFF–9131-00069B).
Supported by the Danish Research Council (DFF–8021-002498, DFF–9131-00069B).
Partially supported by a research grant (VIL51463) from VILLUM FONDEN. Part of this work was done while the author was visiting the Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria.

Authors' address: P. Bille, I. L. Gørtz, and T. A. Steiner, Technical University of Denmark, DTU Compute, Richard Petersens Plads, Lyngby, Denmark; emails: {phbi, inge, terst}@dtu.dk.

1549-6325/2023/09-ART32 $15.00
https://doi.org/10.1145/3607141

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.

We focus on the classic **Lempel-Ziv 1977 (**LZ77**)** [44] 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).

While the opposite problem, where the indexed string S is compressed and the pattern P is uncompressed, is well studied [3, 4, 9, 11–14, 17, 23–25, 31, 32, 35–38, 41] (see also the surveys [19, 39–41]), little is known about the string indexing with compressed pattern problem. As an intermediate result in their paper on indexed multi-pattern matching, Gagie et al. [18] give a data structure using $nH_k(S) + o(n(H_k(S) + 1))$ bits, which can find the suffix array interval for an LZ77-compressed pattern in $O(z \log^2 m \log^{1+\epsilon} n)$ time, where z is the number of phrases in the LZ77 compression of the pattern. Their strategy is to convert the LZ77 compression to a **straight-line program (SLP)** and use iterative merging of suffix array intervals for concatenated strings. Combined with the more recent data structure by Fischer et al. [15], this implies a solution to the string indexing with compressed pattern problem using linear space and $O(z \log(m/z) \log\log n + \text{occ})$ query time. However, since these solutions convert the LZ77 compression to an SLP, the size of the SLP compression is a bottleneck for the query time. The best-known conversion achieves an SLP of size $O(z \log(m/z))$ and the size of the smallest SLP is lower bounded by $\Omega(z \log m/\log\log m)$ [8].

We present a new solution to the string indexing with compressed pattern problem achieving the following bound.

Theorem 1.1. *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 1.1 is optimal within an additive $O(\log m)$ term. In the common case when $z = \Omega(\log m)$ 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 1.1, 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 in the length of the string 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.

Let z_i be the number of phrases in the LZ77 compression of the ith suffix of S, for all $0 \leq i \leq n-1$. We show how to build the data structure from Theorem 1.1 in $O(n \log n + \sum_{i=0}^{n-1} z_i \log\log n)$ expected time for LZ77 without self-references, and $O(n^2 + \sum_{i=0}^{n-1} z_i \log\log n)$ expected time if we allow self-referencing. Further, we show how to extend our results to the **Lempel-Ziv 1978 (LZ78)** [45] compression scheme in the same complexities.

A related problem has been studied in a line of work on *fully compressed pattern matching*, where the goal is to locate a pattern P within a string S when both are given in compressed form [21, 22, 27–29].

The article 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. In Sections 5 and 6, we show how to improve the $\log n$ term to $\log m$, proving the main theorem. In Section 7, we show how to extend these results to the LZ78 compression scheme.

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 ith *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 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 [26, 43].

Compact Tries. Let D be a set of strings $S^1,\dots,S^l$, and assume without loss of generality that the strings in D are prefix-free (if they are not, append each string with a special character \$ that is not in the alphabet). A *compact trie* for D 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. 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 $O(\sum_{i=1}^{l}|S^i|)$. 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 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 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 [43]. 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 is 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. If $l_i > r_i$, we call f_i *self-referencing*. 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 [42];

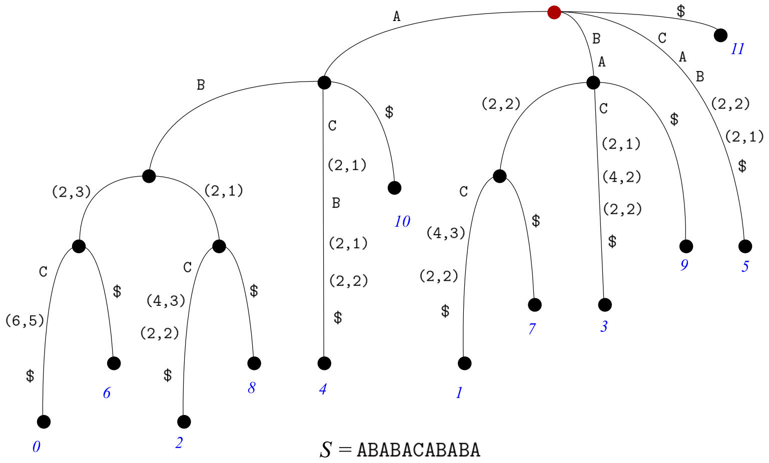


Fig. 1. The phrase trie for the string ABABACABABA\$. In this example, the leaves are sorted according to the lexicographic order of the original suffixes. For instance, the 6th suffix ABABA\$ has the LZ77 parse `A B (2,3) $`, and this string corresponds to the concatenation of labels on the path from the root to the second leaf.

the original one [44] adds a character to each phrase so that it outputs triples instead of tuples.) We have $z = O(n/\log_\sigma n)$. Furthermore, if self-references are not allowed, then $z = \Omega(\log n)$, whereas $z = \Omega(1)$ for self-referential parses.

Every LZ77-compressed string is a string over the extended alphabet that 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.

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 \$ that 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 two children, the number of nodes and edges is $O(n)$. However, we have to store labels corresponding to the LZ77 parses of the suffixes of S, using worst case $\Theta(n^2)$ space.

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. Now, the problem reduces to the following: given the set of suffixes for which the LZ77 compression maximally agrees with P, assuming they match for $k - 1$ phrases and until position p, find the subset of those suffixes for which the kth phrase agrees longest with the kth phrase of P. For a fixed string S of length n, there are at most n different choices for position p, and at most n^2 different choices for the encoding of the next phrase in the pattern. As such, we can store the precomputed solutions for all cases in a table using an additional $O(n^3)$ space for solving the problem in $O(z + \text{occ})$ time. Instead, we will store a linear space and constant time lcp data structure for suffixes of 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 suffixes of S. This will allow us to search for the longest match of P in S in at most $O(\log n)$ extra time.

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 3.1. *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 $f_1 = f'_1, f_2 = f'_2, \ldots, f_{k-1} = f'_{k-1}$ for some k. 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:*

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

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

Proof. To prove the lower bound stated in Equation (1), we will show by induction that for any $i \leq \min(\texttt{lcp}(S\,[p_k,\ n]\,, S[p_k - r'_k,\ n]), l'_k)$, 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

$$\begin{aligned} S'[p_k + i - 1] &= S'[p_k - r'_k + i - 1] && (2)\\ &= S[p_k - r'_k + i - 1] && (3)\\ &= S[p_k + i - 1], && (4) \end{aligned}$$

where Equation (2) follows from $i \leq l'_k$ and because $p_k - r'_k$ is the source of phrase f'_k, Equation (3) follows from the induction hypothesis, and Equation (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 $t = \min(\texttt{lcp}(S\,[p_k,\ n]\,, S[p_k - r'_k,\ n]), l'_k)$. 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 Equation (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 Equation (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 Equation (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.

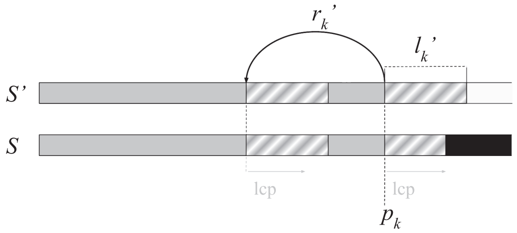


Fig. 2. The kth phrase in S' is copied from position $p_k - r_k'$, at which point S and S' are identical; the `lcp` value gives how far p_k and $p_k - r_k'$ match in S.

3.2 Algorithm

The algorithm we describe in this section, as well as all solutions presented later in the article, actually solve the more general problem of finding all occurrences of the longest prefix of P that is a substring of S. We begin by matching $\mathrm{LZ}\,(P)$ as far as possible in the phrase trie. Let $v = \texttt{locus}(\mathrm{LZ}\,(P))$. Let k be the first phrase in $\mathrm{LZ}\,(P)$ that does not match any of the next phrases in the trie. If v is a node, then 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, in the following way. For the suffix S_i that corresponds to the index in the middle of the given range, 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, let $t = p_k + \texttt{lcp}(S\,[i + p_k,\ n]\,, S\,[i + p_k - r_k,\ n]) + 1$. We compare $S_i[t] = S[i+t]$ with $P[t] = S_i[t-r_k] = S[i+t-r_k]$. If $P[t]$ is lexicographically smaller, we recurse on the left half of the current range, otherwise on the right. Throughout the process, we keep track of the longest match we have seen so far, since updating the search does not necessarily mean that a longer match can be found in the new interval. At the end of the search, we go to the longest match seen and check left and right for all occurrences, since there can be some that the binary search skipped: Multiple consecutive leaves can share the same longest prefix with P while some of them are lexicographically smaller and some are lexicographically bigger.

3.3 Correctness

The compact trie gives us the longest matching prefix of $\mathrm{LZ}\,(P) = f_1 \ldots f_{z_p}$ in the phrase trie. That is, we find all suffixes $S_i = f_1' \cdots f_{z_i}'$ for $i = 0, \ldots, n-1$ such that $f_1 = f_1', \ldots, f_{k-1} = f_{k-1}'$ 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 of 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 3.1 and since $S_i[p] = S[p+i]$ for any p, we have that

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

As such, we return the correct length, and since the match ends on this edge the occurrences of the longest prefix of P correspond to the suffix array interval stored at the next node below.

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

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

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: At any point in the search, when we compare with a suffix S_i and if there exists another suffix that has a longer common prefix with P, it will be lexicographically smaller than suffix S_i exactly if P is lexicographically smaller than S_i. If we compare with a suffix S_i that maximally shares a prefix with P, there might be other suffixes both lexicographically bigger and smaller than S_i that share the same prefix. However, they will all be adjacent, and by checking the adjacent positions of the longest match 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 $O(\sum_{i=0}^{n-1} z_i) = O(n^2)$ 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 3.2. *The phrase trie solves the string indexing with compressed pattern problem in* $O(n^2)$ *space and* $O(z + \log n + \text{occ})$ *time.*

3.5 Preprocessing

The suffix tree and suffix array can be constructed in time $O(\text{sort}(n, \sigma))$, where σ is the size of the alphabet and $\text{sort}(n, \sigma)$ is the sorting complexity of sorting n numbers from a universe of size σ. This is linear in n for linear-sized alphabets [10, 30, 43]. To enable constant time access of the correct outgoing edge at each node, we use perfect hashing [16], which requires an additional $O(n)$ expected preprocessing time. The NCA data structure used for the `lcp` data structure can be constructed in time $O(n)$ [26].

To construct the phrase trie, we need to find the LZ77 parses of all the suffixes. To compute the LZ77 compression of each suffix, we will use results by Keller et al. [34] for generalized substring compression. The data structure in [34] uses a suffix tree augmented with constant amount of information per node, which can be constructed in linear time, plus an NCA data structure and a range successor data structure, to compress any substring $S[i, j]$ of S in time $O(z_{i,j} \cdot Q_{\text{succ}})$. Here, $z_{i,j}$ is the number of phrases in $S[i, j]$ and Q_{succ} is the query time for range successor. An $O(n \log \log n)$ space and $O(\log \log n)$ time range successor data structure can be built in $O(n\sqrt{\log n})$ time [20].

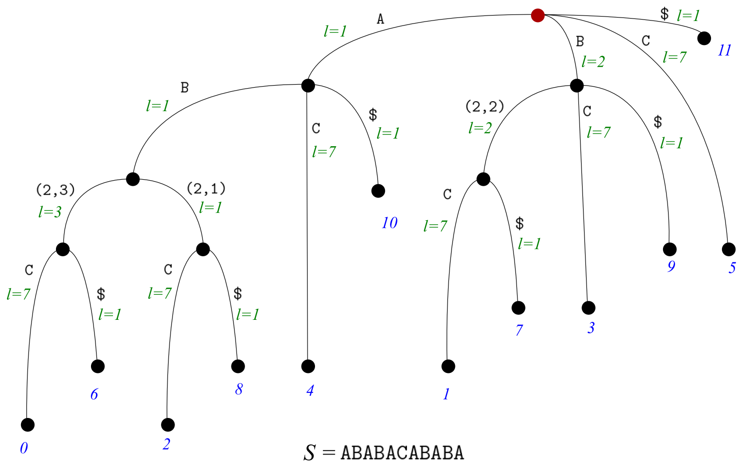


Fig. 3. The phrase trie for the string ABABACABABA using linear space.

Thus, we can find the LZ77 parses of all suffixes in time $O(n\sqrt{\log n}+\sum_{i=0}^{n-1} z_i \log\log n)$. Since we already constructed the suffix tree, we can assume we have the LZ77 parses of all suffixes sorted by lexicographic order. Using perfect hashing again, we can build the phrase trie from those in $O(n+\sum_{i=0}^{n-1} z_i)$ expected time.

Summing up, we can build the data structure in $O(n\sqrt{\log n}+\sum_{i=0}^{n-1} z_i \log\log n)$ expected 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 3.1 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. In contrast to the usual blind trie, we do not store the actual strings or the compressed strings on the side to verify, but show that the structure of the LZ77-compression scheme is enough to ensure navigating within the blind trie without false positives. 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 [16], 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 suffixes S.

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.

- If it is not, we proceed exactly as in the previous section in the case where the locus is at a node.
- 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 in P 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 together with the interval of occurrences stored at the next node below.
- The kth phrase in P is represented by (r_k, l_k):
  - If $\min(\texttt{lcp}(S[i + p_k,\ n], S[i + p_k - r_k,\ n]), l_k) = l_k$, 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 3.1, 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 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 lcp queries. In summary, this gives the following lemma.

Lemma 4.1. *We can solve the string indexing with compressed pattern problem in $O(n)$ space and $O(z + \log n + \text{occ})$ time.*

5 SLICE TREE SOLUTION

In this section, we show how to reduce the $O(\log n)$ time overhead to $O(\log m)$. This will originally give a space complexity of $O(n \log n)$. In the next section, we show how to reduce the space to linear. 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 $O(z + \text{occ})$ 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.

We note that similar results follow from using *z-fast tries* [2], however, we present a direct and simple 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* [33] 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 S, S', and S'' where $S = S'S''$, given the fingerprint of any two of those strings and constant additional information, the third one can be computed in constant time.

Lemma 5.1. *Let S, S', and S'' be three strings satisfying $S = S'S''$. Given the Karp-Rabin fingerprints ϕ of any two of the strings S, S', and S'', we can calculate the third one as follows:*

$$\phi(S) = \phi(S') + x^{|S'|} \cdot \phi(S'') \mod p,$$
$$\phi(S') = \phi(S) - \frac{x^{|S|}}{x^{|S''|}} \cdot \phi(S'') \mod p,$$
$$\phi(S'') = \frac{\phi(S) - \phi(S')}{x^{|S'|}} \mod p.$$

It follows that given the fingerprints of all suffixes of a string S as well as the exponents x^j mod p for all $j = 0, \ldots, n-1$, 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 the paragraph on preprocessing. 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* with respect to a parameter χ: Every vertex v of minimal depth with no more than χ leaves below it is the root of a bottom tree that 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.

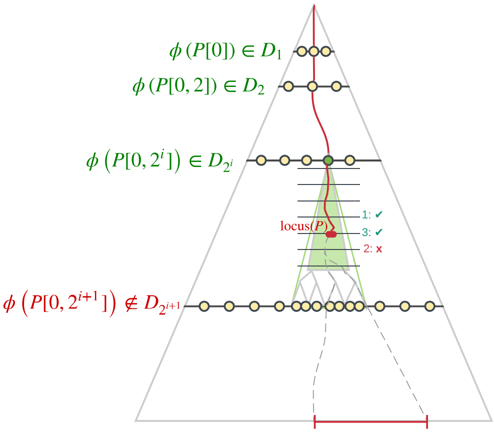
Lemma 5.2 (Alstrup et al. [1]). *The ART decomposition with parameter χ for a rooted tree T with n leaves produces a top tree with at most $\frac{n}{\chi+1}$ leaves.*

5.1 The Slice Tree Decomposition

The data structure we define in this section is only used for speeding up the matching of the kth phrase, assuming k is the maximum number such that we matched $k-1$ phrases in the phrase trie. The overall idea is to use fingerprints to do an exponential search for the locus of P; since the locus is of depth at most m, this way, we will achieve a search time of $O(\log m)$. In order to do this, we will store fingerprints of substrings of S for lengths of powers of 2. For the detailed search of the remainder, we divide the suffix tree into smaller trees, the *slice trees*, where the heights are powers of 2 and increase with the depth in the tree. In order not to use too much space, we store an ART decomposition of the slice trees. This way we can afford to store fingerprints for every string depth in the top tree and binary search within the bottom trees. This will result in an $O(n \log n)$ space data structure. However, in the next section we show how to reduce this space to linear.

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 2. 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.



$\phi\,(P[0]) \in D_1$
$\phi\,(P[0,2]) \in D_2$
$\phi\left(P[0,2^i]\right) \in D_{2^i}$
locus(P)
1: ✓
3: ✓
2: x
$\phi\left(P[0,2^{i+1}]\right) \notin D_{2^{i+1}}$

Fig. 4. Matching in the slice tree: First, we find the lowest i such that the fingerprint of a prefix of P is present at level 2^i. Then we go to the corresponding slice tree and binary search for fingerprints within the top tree.

- 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.

5.2 Algorithm

To match P, we first match the full phrases in the phrase trie until we find the first phrase f_k that 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). For now, we assume $l_k \leq r_k$. At the end of the section, we explain how to deal with self-referencing phrases. 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 locus of LZ (P) 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 two levels in the following way:
  - For $j \in \{2^{\lceil \log p_k \rceil} - p_k, 2^{\lceil \log p_k \rceil + 1} - p_k, \ldots, 2^{\lfloor \log n \rfloor} - p_k\}$ and while $j < l_k$, we find the fingerprint of the prefix $f_k\,[0,\ j] = S\,[i_0 + p_k - r_k,\ i_0 + p_k - r_k + j]$ 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 and 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 a binary 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 (Figure 4). 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. 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 any representative leaf below each bottom tree leaf. For each bottom tree leaf that has a longest match with P, report all suffix tree leaves below.

We note that the search algorithm is similar to prefix search in a z-fast trie; however, there are subtle differences. Let the *2-fattest number* of an interval $[l, r]$ be the unique number of the form $b2^i$ in the interval such that b is an integer and i is maximal. If $l = 0$, this is the largest power of 2 that is at most r. The search in a z-fast trie begins with computing the 2-fattest number in $[0, m - 1]$, finds the first node in the trie that shares a prefix of that length with P (if it exists), and then continues similarly to a binary search. The interval lengths in that search depend on the depths of the nodes and might not be powers of 2, which is why the search relies on finding 2-fattest numbers. In contrast, our search can be seen as an exponential search for the length p of the longest prefix of P in the trie. That is, first, we find in $O(\log p) = O(\log m)$ steps the 2-fattest number in $[0, p]$; call this q. Then, we binary search for p in $[q, 2q]$, where the interval lengths are powers of 2.

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 search in the power of two levels in the suffix tree. We know that if $P\left[0,\ p_k + j\right] = S\left[i,\ i + p_k + j\right]$ for some i, then $\phi(P\left[0,\ p_k\right]) + x^{p_k}\phi(f_k\left[0,\ j\right]) \mod p$ 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, and (since we assume $l_k \leq r_k$) we know that both $P\left[0,\ p_k\right]$ and f_k are substrings of S. Thus, by checking $\phi(P\left[0,\ p_k\right]) = \phi(S\left[i,\ i + p_k\right])$ separately, Lemma 5.1 implies that $P\left[0,\ p_k + j\right]$ and $S\left[i,\ i + p_k + j\right]$ are actually identical. Together, this means that by finding the biggest j such that $p_k + j$ is a power of 2 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 binary search in the top tree is the same as for the search on the power of 2 levels. When we end the binary search, we found the position in the top tree of maximum depth that corresponds to a substring of S matching a prefix of P. 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 within 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 2 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 5.2, 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 $\sum_{T \text{ is slice tree}} O\left(|T|\right) = O(\texttt{\#nodes in suffix tree + extra nodes}) = O(n \log n)$.

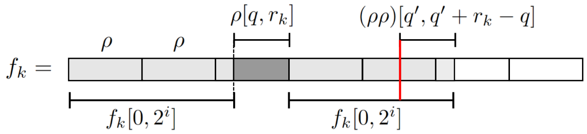


Fig. 5. The prefix of length 2^{i+1} can be constructed from the prefix of length 2^i and substrings of ρ (respectively, $\rho\rho$).

For the time complexity, as before, we use $O(z)$ time 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 $h \leq m$. It follows that the binary 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 $h \leq m$ 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 $O(z + \log m + \text{occ})$.

5.5 Handling Self-Referencing Phrases

Now we describe how to use the slice tree for matching phrase f_k in the case that f_k is self-referencing. Assume we already matched the first $k-1$ phrases in the phrase trie. We will show how to match f_k in the slice tree in three steps: First, we will show how to construct the fingerprints of all prefixes of f_k of length a power of 2 in $O(\log m)$ time. Then we show how this information enables us to find the longest match in the slice tree in $O(\log m)$ time. Finally, we will show how to check for false positives.

Let $f_k = (r_k, l_k)$ with $l_k > r_k$ and let p_k denote the starting position of f_k in P. Since we have matched P up to position p_k, the first r_k letters of f_k are given by a substring of $S[j+p_k-r_k, j+p_k]$, where j is any leaf below the last position we matched in the phrase trie. Call this substring ρ. Since f_k is self-referencing, it is periodic, that is, it is a concatenation of copies of ρ (where the last one might be incomplete).

Finding all Fingerprints of Power of 2 Prefixes. First, we find and store the fingerprints of all prefixes of f_k where the length is a power of 2, by repeatedly doubling and using Lemma 5.1.

While $2^i \leq r_k$, $f_k[0, 2^i]$ is a substring of $S[j + p_k - r_k, j + p_k]$ and we can find its fingerprint in constant time. Then, given the fingerprint of $f_k[0, 2^i]$, while $2^{i+1} \leq l_k$, we can find the fingerprint of $f_k[0, 2^{i+1}]$ in constant time: Note that $f_k[0, 2^{i+1}]$ is a concatenation of $f_k[0, 2^i]$, some suffix of ρ to "fill up" until the end of the next period, and a prefix of $f_k[0, 2^i]$ (see Figure 5). This last prefix can be constructed from $f_k[0, 2^i]$ by subtracting a substring of $\rho\rho$. Thus, the fingerprints of $f_k[0, 2^{i+1}]$ can be computed by combining the fingerprints of these substrings. More precisely, let $q = 2^i \mod r_k$. That means, at the end of $f_k[0, 2^i]$ there is a period cut off after q characters. So if we concatenate $f_k[0, 2^i]$ and $\rho[q, r_k]$, we get a prefix of f_k that consists of full periods only. If we then append $f_k[0, 2^i]$, we get a periodic string of length $2^{i+1} + r_k - q$, so we need to "cut off" the last $r_k - q$ elements. Let $q' = 2^{i+1} \mod r_k$. The substring we cut off corresponds to $(\rho\rho)[q', q' + r_k - q]$.

By assumption, we know the fingerprint of $f_k[0, 2^i]$. We can find the fingerprints of substrings of ρ in constant time by translation to S, and substrings of $\rho\rho$ as a concatenation of at most two substrings of ρ. Using Lemma 5.1, we can thus find the fingerprint of $f_k[0, 2^{i+1}]$ in constant time.

The total time for finding the fingerprints of all prefixes of f_k of length a power of 2 thus takes $O(\log m)$ time.

Matching in the Slice Tree. Once we have stored all fingerprints of prefixes of f_k of length a power of 2, we can find the fingerprint of *any* substring of f_k that is a power of 2 *in constant time.* Let $s = f_k[a, 2^i + a]$ be such a substring. There are two cases:

- If the substring starts at a position $j \cdot r_k$ for some j, then due to the periodicity $f_k[a, 2^i + a] = f_k[0, 2^i]$, and thus it has the same fingerprint as the prefix of the same length.
- Otherwise, due to the periodicity of f_k the substring is equal to some $f_k[b, 2^i + b]$ for $b < r_k$. Now $f_k[b, 2^i + b]$ can be constructed by concatenating the suffix $f_k[b, 2^i]$ of $f_k[0, 2^i]$ with a substring of $\rho\rho$, and thus we can compute its fingerprint in constant time as described above.

Now, we can match in the slice tree in the following way: We find the fingerprint of the prefix of f_k from its starting position in the suffix tree to the next power of 2 level. Since this is a concatenation of at most $\log m$ substrings of length that are a power of 2, we can do this in $O(\log m)$ time. Then, we can "jump" between power of 2 levels in additional constant time. That is, we only need constant time for each step in the exponential search over the power of 2 levels, because we only need to add or subtract the fingerprint of a substring that is a power of 2. Similarly, when binary searching within a slice tree, we always make steps of length that are a power of 2, hence, every step can be done in constant time. Thus, just as in the case for non-self-referencing phrases, the fingerprint search takes a total of $O(\log m)$ time.

Checking for False Positives. Having found the longest match of fingerprints within the slice tree, we can check for false positives in $O(\log m)$ time, by a similar repeated doubling trick as before. Let j be a leaf below the last matched position in the slice tree. We will iteratively check if prefixes of power of 2 lengths of f_k are actually substrings of S and match the corresponding positions in S_j. At every step, we check if the partial fingerprints match the fingerprints of corresponding substrings of S_j, and use that ϕ is constructed such that distinct substrings of S have different fingerprints.

In detail, let $i = 0, \ldots, \lfloor \log l_k \rfloor$. While $2^i \le r_k$, we know that $f_k[0, 2^i]$ is a substring of S. We check if $\phi(f_k[0, 2^i]) = \phi(S_j[p_k, p_k + 2^i))$. If yes, then the substrings are the same. For $2^i > r_k$, assuming we have verified that $f_k[0, 2^i]$ is a substring of S and is equal to $S_j[p_k, p_k + 2^i]$, we can check if $f_k[0, 2^{i+1}]$ is equal to $S_j[p_k, p_k + 2^{i+1}]$ in the following way: Since $f_k[0, 2^{i+1}] = f_k[0, 2^i]\rho[q, r_k]f_k[0, 2^i - (r_k - q)]$, we know that all substrings on the right are substrings of S. Additionally, we know that $f_k[0, 2^i] = S_j[p_k, p_k + 2^i]$. We check if $\phi(\rho[q, r_k]) = \phi(S_j[p_k + 2^i, p_k + 2^i + r_k - q])$ and if $\phi(f_k[0, 2^i - (r_k - q)]) = \phi(S_j[p_k + 2^i + r_k - q, 2^{i+1}])$. We can compute all these fingerprints in constant time, and since we always compare fingerprints of substrings of S, we know that if the fingerprints are the same, then the substrings are the same. Hence, in that case, $f_k[0, 2^{i+1}]$ is equal to $S_j[p_k, p_k + 2^{i+1}]$, which also means it is a substring of S. After we have verified the prefix $f_k[0, 2^{\lfloor \log l_k \rfloor}]$, the full f_k is again a concatenation of $O(\log m)$ strings we have already verified to be substrings of S, and we can check them in the same way.

We arrive at the following result.

Lemma 5.3. *The slice tree solution solves the string indexing with compressed pattern problem in $O(n \log n)$ space and $O(z + \log m + \mathrm{occ})$ time.*

5.6 Preprocessing

We now describe how to construct the data structure, especially, how to choose the fingerprint function $\phi_{p,x}$.

Choosing $\phi_{p,x}$. In [33], it is shown that for good choice of p and uniformly random $x \in \mathbb{Z}_p$, the fingerprint function $\phi_{p,x}$ is collision-free on substrings of S with high probability.

For any choice of x, we can check if $\phi_{p,x}$ is collision-free in expected time $O(n^2)$ and $O(n)$ additional space. Since in the algorithm, we only ever compare the fingerprints of substrings that have the same length, it is enough to make sure $\phi_{p,x}$ is collision-free on substrings of a given length l (we can also trivially extend such a fingerprint to a fingerprint function that is collision-free on all substrings of S, but we don't need to). Now, for every $1 \leq l \leq n - l$, we simply compute all $\phi_{p,x}(S[i, i + l])$ for $0 \leq i \leq n - 1$ and keep a dictionary using universal hashing [7], to check if any two substrings of length l map to the same fingerprint. Then we discard the dictionary.

Since $\phi_{p,x}$ is collision-free with high probability, we will find a collision-free $\phi_{p,x}$ in expected constant number of tries, which gives an expected $O(n^2)$ running time for finding a collision-free $\phi_{p,x}$.

Faster Construction for LZ77 *without Self-References.* If we use LZ77 without self-referencing, we can match f_k in the phrase trie and then check for false positives in $\log m$ time using only fingerprints that are a power of 2 long, in the following way: We can divide f_k into $\log m$ substrings of lengths that are a power of 2. Since f_k is not self-referencing, all these are substrings of S. We compare their fingerprints with the fingerprints of the corresponding substrings in the potential match. Thus, in the case of non-self referencing, it is enough that $\phi_{p,x}$ is collision-free on substrings of S that have a length that is a power of 2, and by the same strategy as before, such a $\phi_{p,x}$ can be constructed in $O(n \log n)$ expected time (see also Bille et al. [5]).

Final Construction. Once we have chosen $\phi_{p,x}$, we precompute the fingerprints of all suffixes of S, which can be done in $O(n)$ time: First, we compute $x^i \mod p$ for all $0 \leq i < n$ in $O(n)$ time, then we use Lemma 5.1 to compute $S_{n-1}, S_{n-2}, \ldots, S_0 = S$, in that order. After we have stored the fingerprints of all suffixes of S, computing the fingerprint of any substring can be done in constant time. Using perfect hashing [16], we can build all dictionaries in expected time linear to their size, in total, $O(n \log n)$ expected time. The ART decompositions can be constructed in time linear in the nodes, i.e., $O(n \log n)$ total worst case time. Together with the preprocessing time from Sections 3 and 4, we can construct the full data structure in

- $O(n^2 + \sum_{i=0}^{n-1} z_i \log\log n)$ expected time if we allow self-referencing;
- $O(n \log n + \sum_{i=0}^{n-1} z_i \log\log n)$ expected time if we do not allow self-referencing.

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 5.3.

6.1 The Data Structure

We start with the slice tree solution. Call every suffix tree 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 the same way 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:

- 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.
- 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 1.1.

7 LZ78-COMPRESSED PATTERNS

As an extension to our result, we show that a very similar solution solves the problem for the LZ78compression scheme [45].

LZ78. Given an input string S of length n, the LZ78 parsing divides S into z substrings $f_1, f_2, \ldots, f_z$, 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 is equal to a phrase f_j, $j < i$, plus the next character. Note that this choice is unique. To compress S, we can then replace each phrase f_i of type (b) with a pair (j, α) such that j is the index of the phrase f_j, and α is the next character.

Data Structure. The phrase trie with respect to LZ78 is defined completely analogously to Section 3; the only difference is that the suffixes of S are LZ78 compressed. The representation from Section 4 can be applied directly. The final data structure consists of the efficient representation of the LZ78 phrase trie together with the (unchanged) slice tree solution defined in Sections 5 and 6.

Algorithm. When matching in the phrase trie, we build up a dictionary mapping LZ78 phrases to substrings in S. That is, assume p_k is the starting position of f_k in P, and we fully matched up until the end of f_k in the phrase trie. Then, we add an entry to the dictionary where the key is the phrase index k and the value is a pair $(j + p_k, j + p_k + |f_k|)$, where j is a leaf below the current position in the phrase trie.

Assume we have matched up to a position p_k of P. Let $f_k = (f_a, \alpha)$ be a new phrase in P and $f'_k = (f'_b, \beta)$ be a new phrase in the phrase trie, and let (i, l) be the dictionary entry at a. That is, i is a starting position of f_a, and l is the length of f_a. Further, let j be a leaf below the current position in the phrase trie. Then, similarly as in Lemma 3.1, we have

$$\texttt{lcp}(P, S[j,n]) \geq p_k + \min(\texttt{lcp}(S[i,n], S[j+p_k, n]), l), \tag{5}$$

$$\texttt{lcp}(P, S[j,n]) = p_k + \min(\texttt{lcp}(S[i,n], S[j+p_k, n]), l), \text{ if } f_a \neq f'_b. \tag{6}$$

To see that Equation (5) is true, note that by definition of i and l, $f_a = S(i, i+l)$. If $f_a \neq f'_b$, then Equation (6) holds by the greedy parsing.

Thus, the main property we need for matching within the (blind) phrase trie is preserved. Note that unlike the version of LZ77 we use in this article, an LZ78 phrase always includes an extra letter at the end. However, that is not an issue, since we can always access the next character in the phrase trie in constant time through S. For the slice tree solution, we only need that the last phrase is encoded as a substring of S, which is given by the dictionary. Thus, all results from the previous sections generalize to LZ78.

We arrive at the following result.

Theorem 7.1. *We can solve the string indexing with compressed pattern problem for* LZ78*-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* LZ78 *compressed pattern.*

8 OPEN PROBLEMS

We have introduced the string indexing with compressed pattern problem and provided a solution achieving almost optimal bounds for LZ77 compressed patterns. Further, we have shown that the results extend to the LZ78 compression scheme. At the same time, these results open some interesting directions for further research:

- Our results are optimal for the LZ77 variant without self-referencing. An interesting open question is if there is a way to get optimal time for the self-referencing variant, that is, get rid of the additional $O(\log m)$ time overhead.
- Similarly, it would be interesting to see if we can get rid of the $O(n^2)$ expected construction time for self-referencing while still giving a Las Vegas algorithm.
- It would be interesting to consider the string indexing with compressed pattern problem for other compression schemes. Specifically, is there a way to compress multiple patterns that allows a similar tradeoff?
- Finally, the related problem where the indexing string and the pattern are *both* compressed is especially interesting for practical use cases, because in many practical scenarios, the indexing string will be very long.

REFERENCES

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

[2] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2010. Dynamic Z-fast tries. In *Proc. 17th SPIRE*. 159–172.

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

[4] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. 2018. Time–space trade-offs for Lempel–Ziv compressed indexing. *Theor. Comput. Sci.* 713 (2018), 66–77.

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

[6] Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. 2020. String indexing with compressed patterns. In *Proc. 37th STACS*. 10:1–10:13.

[7] Larry Carter and Mark N. Wegman. 1977. Universal classes of hash functions (extended abstract). In *Proc. 9th STOC*. 106–112.

[8] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Trans. Inf. Theory* 51, 7 (2005), 2554–2576.

[9] Francisco Claude and Gonzalo Navarro. 2012. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*. 180–192.

[10] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. 2000. On the sorting-complexity of suffix tree construction. *J. ACM* 47, 6 (2000), 987–1011.

[11] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *Proc. 41st FOCS*. 390–398.

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

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

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

[15] Johannes Fischer, Dominik Köppl, and Florian Kurpicz. 2016. On the benefit of merging suffix array intervals for parallel pattern matching. In *Proc. 27th CPM*. 26:1–26:11.

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

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

[18] Travis Gagie, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, Leena Salmela, and Jorma Tarhio. 2012. Indexed multi-pattern matching. In *Proc. 10th LATIN*. 399–407.

[19] Travis Gagie and Simon J. Puglisi. 2015. Searching and indexing genomic databases via kernelization. *Front. Bioeng. Biotechnol.* 3 (2015), 12.

[20] Younan Gao, Meng He, and Yakov Nekrich. 2020. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In *Proc. 28th ESA*. 54:1–54:18.

[21] Leszek Gasieniec and Wojciech Rytter. 1999. Almost optimal fully LZW-compressed pattern matching. In *Proc. 9th DCC*. 316–325.

[22] Pawel Gawrychowski. 2012. Tying up the loose ends in fully LZW-compressed pattern matching. In *Proc. 29th STACS*. 624–635.

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

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

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

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

[27] Masahiro Hirao, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. 2000. Fully compressed pattern matching algorithm for balanced straight-line programs. In *Proc. 7th SPIRE*. 132–138.

[28] Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. 2005. A fully compressed pattern matching algorithm for simple collage systems. *Int. J. Found. Comput. Sci.* 16, 6 (2005), 1155–1166.

[29] Artur Jez. 2015. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms* 11, 3 (2015), 20:1–20:43.

[30] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936.

[31] Juha Kärkkäinen and Erkki Sutinen. 1998. Lempel-Ziv index for q-Grams. *Algorithmica* 21, 1 (1998), 137–154.

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

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

[34] Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. 2014. Generalized substring compression. *Theor. Comput. Sci.* 525 (2014), 42–54.

[35] Sebastian Kreft and Gonzalo Navarro. 2013. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.* 483 (2013), 115–133.

[36] Veli Mäkinen. 2000. Compact suffix array. In *Proc. 11th CPM*. 305–319.

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

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

[39] Gonzalo Navarro. 2012. Indexing highly repetitive collections. In *Proc. 23rd IWOCA*. 274–279.

[40] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach*. Cambridge University Press.

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

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

[43] Peter Weiner. 1973. Linear pattern matching algorithms. In *Proc. 14th FOCS*. 1–11.

[44] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343.

[45] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536.

Received 19 March 2020; revised 3 February 2022; accepted 15 June 2023