Time-space trade-offs for lempel-ziv compressed indexing

Bille, Philip; Ettienne, Mikko Berggren; Gørtz, Inge Li; Vildhøj, Hjalte Wedel

Published in:
Proceedings of 28th Annual Symposium on Combinatorial Pattern Matching

Link to article, DOI:
10.4230/LIPIcs.CPM.2017.16

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Time-Space Trade-Offs for Lempel-Ziv Compressed Indexing

Philip Bille[*1], **Mikko Berggren Ettienne**[†2], **Inge Li Gørtz**[‡3], **and Hjalte Wedel Vildhøj**[4]

1 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
phbi@dtu.dk
2 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
miet@dtu.dk
3 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
inge@dtu.dk
4 **Technical University of Denmark, DTU Compute, Lyngby, Denmark**
hwvi@dtu.dk

Abstract

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*. The goal is to use little space relative to the compressed size of S while supporting fast queries. We present a compressed index based on the Lempel-Ziv 1977 compression scheme. Let n, and z denote the size of the input string, and the compressed LZ77 string, respectively. We obtain the following time-space trade-offs. Given a pattern string P of length m, we can solve the problem in

(i) $O(m + \text{occ} \lg \lg n)$ time using $O(z \lg(n/z) \lg \lg z)$ space, or
(ii) $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \text{occ}(\lg \lg n + \lg^\epsilon z))$ time using $O(z \lg(n/z))$ space, for any $0 < \epsilon < 1$

In particular, (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by a factor $\lg \lg z$. Alternatively, (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$. Our index also supports extraction of any substring of length ℓ in $O(\ell + \lg(n/z))$ time. Technically, our results are obtained by novel extensions and combinations of existing data structures of independent interest, including a new batched variant of weak prefix search.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, E.4 Coding and Information Theory, E.1 Data Structures

Keywords and phrases compressed indexing, pattern matching, LZ77, prefix search

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.16

1 Introduction

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*, that is, given a string P, report all occurrences of substrings in S that match P. Here the compressed representation can be any

* Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
† Supported by the Danish Research Council (DFF – 4005-00267).
‡ Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).

compression scheme or measure (kth order entropy, smallest grammar, Lempel-Ziv, etc.). The goal is to use little space relative to the compressed size of S while supporting fast queries. Compressed indexing is a key computational primitive for querying massive data sets and the area has received significant attention over the last decades with numerous theoretical and practical solutions, see e.g. [25, 12, 29, 23, 13, 14, 21, 22, 15, 34, 30, 9, 27, 18, 24, 4] and the surveys [34, 32, 33, 19].

The Lempel-Ziv 1977 compression scheme (LZ77) [37] is a classic compression scheme based on replacing repetitions by references in a greedy left-to-right order. Numerous variants of LZ77 have been developed and several widely used implementations are available (such as `gzip` [20]). Recently, LZ77 has been shown to be particularly effective at handling highly-repetitive data sets [30, 32, 27, 8, 3] and LZ77 compression is always at least as powerful as any grammar representation [36, 7].

In this paper, we consider compressed indexing based on LZ77 compression. Relatively few results are known for this version of the problem. Let n, z, and m denote the size of the input string, the compressed LZ77 string, and the pattern string, respectively. Kärkkäinen and Ukkonen introduced the problem in 1996 [25] and gave an initial solution that required read-only access to the uncompressed text. Interestingly, this work is among the first results in compressed indexing [34]. More recently, Gagie et al. [17, 18] revisited the problem and gave a solution using space $O(z \lg(n/z))$ and query time $O(m \lg m + \mathrm{occ} \lg \lg n)$, where occ is the number of occurrences of P in S. Note that these bounds assume a constant sized alphabet.

1.1 Our Results

We show the following main result.

▶ **Theorem 1.** *Given a string S of length n from a constant sized alphabet compressed using LZ77 into a string of length z we can build a compressed-index supporting substring queries in:*

(i) $O(m + \mathrm{occ} \lg \lg n)$ *time using* $O(z \lg(n/z) \lg \lg z)$ *space, or*

(ii) $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg \lg n + \lg^\epsilon z))$ *time using* $O(z \lg(n/z))$ *space, for any* $0 < \epsilon < 1$

Compared to the previous bounds Theorem 1 obtains new interesting trade-offs. In particular, Theorem 1 (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by only a factor $\lg \lg z$. Alternatively, Theorem 1 (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$.

Gagie et al. [18] also showed how to extract an arbitrary substring of S of length ℓ in time $O(\ell + \lg n)$. We show how to support the same extraction operation and slightly improve the time to $O(\ell + \lg(n/z))$.

Technically, our results are obtained by new variants and extensions of existing data structures in novel combinations. In particular, we consider a batched variant of the *weak prefix search problem* and give the first non-trivial solution to it. We also generalize the well-known bidirectional compact trie search technique [28] to reduce the number of queries at the cost of increasing space. Finally, we show how to combine this efficiently with range reporting and fast random-access in a balanced grammar leading to the result.

As mentioned all of the above bounds hold for a constant size alphabet. However, Theorem 1 is an instance of full time-space trade-off that also supports general alphabets. We discuss the details in Section 8 and Appendix 8.1.

2 Preliminaries

We assume a standard unit-cost RAM model with word size $w = \Theta(\lg n)$ and that the input is from an integer alphabet $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$ and measure space complexity in words unless otherwise specified.

A string S of length $n = |S|$ is a sequence $S[1] \ldots S[n]$ of n characters drawn from Σ. The string $S[i] \ldots S[j]$ denoted $S[i, j]$ is called a *substring* of S. ϵ is the empty string and $S[i, i] = S[i]$ while $S[i, j] = \epsilon$ when $i > j$. The substrings $S[1, i]$ and $S[j, n]$ are the i^{th} *prefix* and the j^{th} *suffix* of S respectively. The reverse of the string S is denoted $\text{rev}(S) = S[n]S[n-1] \ldots S[1]$.

Let D be a set of k strings and let T_D be the compact trie storing all the strings of D. $\text{str}(v)$ denotes the prefix corresponding to the vertex v. The *depth* of vertex v is the number of edges on the path from v to the root. We assume each string in D is terminated by a special character $\$ \notin \Sigma$ such that each string in D corresponds to a leaf. The children of each vertex are sorted from left to right in increasing lexicographical order, and therefore the left to right order of the leaves corresponds to the lexicographical order of the strings in D. Let $\text{rank}(s)$ denote the rank of the string $s \in D$ in this order. The *skip interval* of a vertex $v \in \text{T}_D$ with parent u is $(|\text{str}(u)|, |\text{str}(v)|]$ denoted $\text{skip}(v)$ and $\text{skip}(v) = \emptyset$ if v is the root. The *locus* of a string s in T_D, denoted $\text{locus}(s)$, is the minimum depth vertex v such that s is a prefix of $\text{str}(v)$. If there is no such vertex, then $\text{locus}(s) = \perp$. In order to reduce the space used by T_D we only store the first character of every edge and in every vertex v we store $|\text{str}(v)|$ (This variation is also known as a PATRICIA tree [31]). We navigate T_D by storing a dictionary in every internal vertex mapping the first character of the label of an edge to the respective child. The size of T_D is $O(k)$.

A *Karp-Rabin fingerprinting function* [26] is a randomized hash function for strings. We use a variation of the original definition appearing in Porat and Porat [35]. The fingerprint for a string S of length n is defined as: $\phi(S) = \sum_{i=1}^{n} S[i] \cdot r^{i-1} \mod p$, where p is a prime and r is a random integer in $\mathbb{Z}_p$ (the field of integers modulo p). Storing the values n, $r^n \mod p$ and $r^{-n} \mod p$ along with a fingerprint allows for efficient composition an subtraction of fingerprints. Using this we can compute and store the fingerprints of each of the prefixes of a string S of length n in $O(n)$ time and space such that we afterwards can compute the fingerprint of any substring $S[i, j]$ in constant time. We say that the fingerprints of the strings x and y *collide* when $\phi(x) = \phi(y)$ and $x \neq y$. A fingerprinting function ϕ is *collision-free* for a set of strings if there are no fingerprint collisions between any of the strings. Porat and Porat [35] show that if x and y are different strings of length at most n and $p = \Theta(n^{2+\alpha})$ for some $\alpha > 0$, then the probability that $\phi(x) = \phi(y)$ is less than $1/n^{1+\alpha}$.

The *LZ77 parse* of a string S of length n is a sequence Z of z subsequent substrings of S called *phrases* such that $S = Z[1]Z[2], \ldots, Z[z]$. Z is constructed in a left to right pass of S: Assume that we have found the sequence $Z[1, i]$ producing the string $S[1, j-1]$ and let $S[j, j'-1]$ be the longest prefix of $S[j, n-1]$ that is also a substring of $S[1, j'-2]$. Then $Z[i+1] = S[j, j']$. The occurrence of $S[j, j'-1]$ in $S[1, j'-2]$ is called the *source* of the phrase $Z[i]$. Thus a phrase is composed by the contents of its possibly empty source and a trailing character which we call the *phrase border* and is typically represented as a triple $Z[i] = (start, len, c)$ where *start* is the starting position of the source, *len* is the length of the source and $c \in \Sigma$ is the border. For a phrase $Z[i] = S[j, j']$ we denote the position of its border by $\text{border}(Z[i]) = j'$ and its source by $\text{source}(Z[i]) = S[j, j'-1]$. For example, the string $abcabcabc \ldots abc$ of length n has the LZ77 parse $|a|b|c|abcabcabc \ldots abc|$ of length 4 which is represented as $Z = (0, 0, a)(0, 0, b)(0, 0, c)(0, n-4, c)$.

3 Prefix Search

The *prefix search* problem is to preprocess a set of strings such that later, we can find all the strings in the set that are prefixed by some query string. Belazzougui et al. [2] consider the *weak prefix search* problem, a relaxation of the prefix search problem where we are only requested to output the ranks (in lexicographic order) of the strings that are prefixed by the query pattern and we only require no false negatives. Thus we may answer arbitrarily when no strings are prefixed by the query pattern.

▶ **Lemma 2** (Belazzougui et al. [2], appendix H.3)**.** *Given a set D of k strings with average length l, from an alphabet of size σ, we can build a data structure using $O(k(\lg l + \lg\lg\sigma))$ bits of space supporting weak prefix search for a pattern P of length m in $O(m\lg\sigma/w + \lg m)$ time where w is the word size.*

The term $m\lg\sigma/w$ stems from preprocessing P with an incremental hash function such that the hash of any substring $P[i,j]$ can be obtained in constant time afterwards. Therefore we can do weak prefix search for h substrings of P in $O(m\lg\sigma/w + h\lg m)$ time. We now describe a data structure that builds on the ideas from Lemma 2 but obtains the following:

▶ **Lemma 3.** *Given a set D of k strings, we can build a data structure taking $O(k)$ space supporting weak prefix search for h substrings of a pattern P of length m in time $O(m + h(m/x + \lg x))$ where x is a positive integer.*

If we know h when building our data structure, we set x to h and obtain a query time of $O(m + h\lg h)$ with Lemma 3.

Before describing our data structure we need the following definition: The *2-fattest* number in a nonempty interval of strictly positive integers is the number in the interval whose binary representation has the highest number of trailing zeroes.

3.1 Data Structure

Let T_D be the compact trie representing the set D of k strings and let x be a positive integer. Denote by $\mathrm{fat}(v)$ the 2-fattest number in the skip interval of a vertex $v \in \mathrm{T}_D$. The *fat prefix* of v is the length $\mathrm{fat}(v)$ prefix of $\mathrm{str}(v)$. Denote by D^{fat} the set of fat prefixes induced by the vertices of T_D. The x-prefix of v is the shortest prefix of $\mathrm{str}(v)$ whose length is a multiple of x and is in the interval $\mathrm{skip}(v)$. If v's skip interval does not span a multiple of x, then v has no x-prefix. Let D^x be the set of x-prefixes induced by the vertices of T_D. The data structure is the compact trie T_D augmented with:

- A fingerprinting function ϕ.
- A dictionary $\mathcal{G}$ mapping the fingerprints of the strings in D^{fat} to their associated vertex.
- A dictionary $\mathcal{H}$ mapping the fingerprints of the strings in D^x to their associated vertex.
- For every vertex $v \in \mathrm{T}_D$ we store the rank in D of the string represented by the leftmost and rightmost leaf in the subtree of v, denoted l_v and r_v respectively.

The data structure is similar to the one by Belazzougui et al. [2] except for the dictionary $\mathcal{H}$, which we use in the first step of our search. There are at most k strings in each of D^{fat} and D^x thus the total space of the data structure is $O(k)$.

Let i be the start of the skip interval of some vertex $v \in \mathrm{T}_D$ and define the *pseudo-fat* numbers of v to be the set of 2-fattest numbers in the intervals $[i,p]$ where $i \le p < \mathrm{fat}(v)$. We require that the fingerprinting function ϕ is collision-free for the strings in D^{fat}, the strings in D^x and all the length l-prefixes of the strings in D where l is a pseudo-fat number in the skip interval of some vertex $v \in \mathrm{T}_D$.

Observe that the range of strings in D that are prefixed by some pattern P of length m is exactly $[l_v, r_v]$ where $v = \text{locus}(P)$. Answering a weak prefix search query for P is comprised by two independent steps. First step is to find a vertex $v \in \mathrm{T}_D$ such that $\text{str}(v)$ is a prefix of P and $m - |\text{str}(v)| \leq x$. We say that v is in *x-range of P*. Next step is to apply a slightly modified version of the search technique from Belazzougui et al. [2] to find the *exit vertex* for P, that is, the deepest vertex $v' \in \mathrm{T}_D$ such that $\text{str}(v')$ is a prefix of P. Having found the exit vertex we can find the locus in constant time as it is either the exit vertex itself or one of its children.

Finding an x-range Vertex. We now describe how to find a vertex in x-range of P. If $m < x$ we simply report that the root of T_D is in x-range of P. Otherwise, let v be the root of T_D and for $i = 1, 2, \ldots \lfloor m/x \rfloor$ we check if $ix > |\text{str}(v)|$ and $\phi(P[1, ix])$ is in $\mathcal{H}$ in which case we update v to be the corresponding vertex. Finally, if $|\text{str}(v)| \geq m$ we report that v is $\text{locus}(P)$ and otherwise we report that v is in x-range of P. In the former case, we report $[l_v, r_v]$ as the range of strings in D prefixed by P. In the latter case we pass on v to the next step of the algorithm.

We now show that the algorithm is correct when P prefixes a string in D. It is easy to verify that the x-prefix of v prefixes P at all time during the execution of the algorithm. Assume that $|\text{str}(v)| \geq m$ by the end of the algorithm. We will show that in that case $v = \text{locus}(P)$, i.e., that v is the highest node prefixed by P. Since P prefixes a string in D, the x-prefix of v prefixes P, and $|\text{str}(v)| \geq m$, then P prefixes v. Since the x-prefix of v prefixes P, P does not prefix the parent of v and thus v is the highest node prefixed by P.

Assume now that $|\text{str}(v)| < m$. We will show that v is in x-range of P. Since P prefixes a string in D and the x-prefix of v prefixes P, then $\text{str}(v)$ prefixes P. Let $P[1, ix]$ be the x-prefix of v. Since v is returned, either $\phi(P[1, jx]) \notin \mathcal{H}$ or $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. If $\phi(P[1, jx]) \notin \mathcal{H}$ then $P[1, jx]$ is not a x-prefix of any node in T_D. Since P prefixes a string in D this implies that jx is in the skip interval of v, i.e., $jx \leq |\text{str}(v)|$. This means that $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. Therefore $\lfloor m/x \rfloor x \leq |\text{str}(v)| < m$ and it follows that $m - |\text{str}(v)| < x$. We already proved that $\text{str}(v)$ prefixes P and therefore v is in x-range of P.

In case P does not prefix any string in D we either report that $v = \text{locus}(P)$ even though $\text{locus}(P) = \perp$ or report that v is in x-range of P because $m - |\text{str}(v)| \leq x$ even though $\text{str}(v)$ is not a prefix of P due to fingerprint collisions. This may lead to a false positive. However, false positives are allowed in the weak prefix search problem.

Given that we can compute the fingerprint of substrings of P in constant time the algorithm uses $O(m/x)$ time.

From x-range to Exit Vertex. We now consider how to find the exit vertex of P hereafter denoted v_e. The algorithm is similar to the one presented in Belazzougui et al. [2] except that we support starting the search from not only the root, but from any ancestor of v_e.

Let v be any ancestor of v_e, let y be the smallest power of two greater than $m - |\text{str}(v)|$ and let z be the largest multiple of y no greater than $|\text{str}(v)|$. The search progresses by iteratively halving the search interval while using $\mathcal{G}$ to maintain a candidate for the exit vertex and to decide in which of the two halves to continue the search.

Let v_c be the candidate for the exit vertex and let l and r be the left and right boundary for our search interval. Initially $v_c = v$, $l = z$ and $r = z + 2y$. When $r - l = 1$, the search terminates and reports v_c. In each iteration, we consider the mid $b = (l + r)/2$ of the interval $[l, r]$ and update the interval to either $[b, r]$ or $[l, b]$. There are three cases:

1. b is out of bounds
   a. If $b > m$ set r to b.
   b. If $b \leq |\text{str}(v_c)|$ set l to b.
2. $P[1,b] \in D^{\text{fat}}$, let u be the corresponding vertex, i.e. $\mathcal{G}(\phi(P[1,b])) = u$.
   a. If $|\text{str}(u)| < m$, set v_c to u and l to b.
   b. If $|\text{str}(u)| \geq m$, report $u = \text{locus}(P)$ and terminate.
3. $P[1,b] \notin D^{\text{fat}}$ and thus $\phi(P[1,b])$ is not in $\mathcal{G}$, set r to b.

Observe that we are guaranteed that all fingerprint comparisons are collision-free in case P prefixes a string in D. This is because the length of the prefix fingerprints we consider are all either 2-fattest or pseudo-fat in the skip interval of locus(P) or one of its ancestors and we use a fingerprinting function that is collision-free for these strings.

Correctness. We now show that the invariant $l \leq |\text{str}(v_c)| \leq |\text{str}(v_e)| < r$ is satisfied and that $\text{str}(v_c)$ is a prefix of P before and after each iteration. After $O(\lg x)$ iterations $r - l = 1$ and thus $l = |\text{str}(v_e)| = |\text{str}(v_c)|$ and therefore $v_c = v_e$. Initially v_c is an ancestor of v_e and thus $\text{str}(v_c)$ is a prefix of P, $l = z \leq |\text{str}(v_c)|$ and $r = z + 2y > m > |\text{str}(v_e)|$ so the invariant is true. Now assume that the invariant is true at the beginning of some iteration and consider the possible cases:

1. b is out of bounds
   a. $b > m$ then because $|\text{str}(v_e)| \leq m$, setting r to b preserves the invariant.
   b. $b \leq |\text{str}(v_c)|$ then setting l to b preserves the invariant.
2. $P[1,b] \in D^{\text{fat}}$, let $u = \mathcal{G}(\phi(P[1,b]))$.
   a. $|\text{str}(u)| \leq m$ then $\text{str}(u)$ is a prefix of P and thus $b = \text{fat}(u) \leq |\text{str}(u)| \leq |\text{str}(v_e)|$ so setting l to b and v_c to u preserves the invariant.
   b. $|\text{str}(u)| \geq m$ yet $u = \mathcal{G}(\phi(P[1,b]))$. Then u is the locus of P.
3. $P[1,b] \notin D^{\text{fat}}$, and thus $\phi(P[1,b])$ is not in $\mathcal{G}$. As we are not in any of the out of bounds cases we have $|\text{str}(v_c)| < b < m$. Thus, either $b > |\text{str}(v_e)|$ and setting r to b preserves the invariant. Otherwise $b \leq |\text{str}(v_e)|$ and thus b must be in the skip interval of some vertex u on the path from v_c to v_e excluding v_c. But $\text{skip}(u)$ is entirely included in (l,r) and because b is 2-fattest in (l,r)[1] it is also 2-fattest in $\text{skip}(u)$. It follows that $\text{fat}(u) = b$ which contradicts $P[1,b] \notin D^{\text{fat}}$ and thus the invariant is preserved.

Thus if P prefixes a string in D we find either the exit vertex v_e or the locus of P. In the former case the locus of P is the child of v_e identified by the character $P[|\text{str}(v')|+1]$. Having found the vertex $u = \text{locus}(P)$ we report $[l_u, r_u]$ as the range of strings in D prefixed by P. In case P does not prefix any strings in D, the fact that the fingerprint of a prefix of P match the fingerprint of some fat prefix in D^x does not guarantee equality of the strings. There are two possible consequences of this. Either the search successfully finds what it believes to be the locus of P even though $\text{locus}(P) = \perp$ in which case we report a false positive. Otherwise, there is no child identified by $P[|\text{str}(v')| + 1]$ in which case we can correctly report that no strings in D are prefixed by S, a true negative. Recall that false positives are allowed as we are considering the weak prefix search problem.

[1] If $b - a = 2^i$, $i > 0$ and a is a multiple of 2^{i-1} then the mid of the interval $(a+b)/2$ is 2-fattest in (a,b).

Complexity. The size of the interval $[l, r]$ is halved in each iteration, thus we do at most $O(\lg(m - |\text{str}(v)|))$ iterations, where v is the vertex from which we start the search. If we use the technique from the previous section to find a starting vertex in x-range of P, we do $O(\lg x)$ iterations. Each iteration takes constant time. Note that if P does not prefix a string in D we may have fingerprint collisions and we may be given a starting vertex v such that $\text{str}(v)$ does not prefix P. This can lead to a false positive, but we still have $m - |\text{str}(v)| \leq x$ and therefore the time complexity remains $O(\lg x)$.

Multiple Substrings. In order to answer weak prefix search queries for h substrings of a pattern P of length m, we first preprocess P in $O(m)$ time such that we can compute the fingerprint of any substring of P in constant time. We can then answer a weak prefix search query for any substring of P in total time $O(m/x + \lg x)$ using the techniques described in the previous sections. The total time is therefore $O(m + h(m/x + \lg x))$.

4 Distinguishing Occurrences

The following sections describe our compressed-index consisting of three independent data structures. One that finds long primary occurrences, one that finds short primary occurrences and one that finds secondary occurrences.

Let Z be the LZ77 parse of length z representing the string S of length n. If $S[i, j]$ is a phrase of Z then any substring of $S[i, j - 1]$ is a *secondary substring* of S. These are the substrings of S that do not contain any phrase borders. On the other hand, a substring $S[i, j]$ is a *primary substring* of S when there is some phrase $S[i', j']$ where $i' \leq i \leq j' \leq j$, these are the substrings that contain one or more phrase borders. Any substring of S is either primary or secondary. A primary substring that match a query pattern P is a *primary occurrence* of P while a secondary substring that match P is a *secondary occurrence* [25].

5 Long Primary Occurrences

For simplicity, we assume that the data structure given in Lemma 3 not only solves the weak prefix problem, but also answers correctly when the query pattern does not prefix any of the indexed strings. Later in Section 5.3 we will see how to lift this assumption. The following data structure and search algorithm is a variation of the classical bidirectional search technique for finding primary occurrences [25].

5.1 Data Structure

For every phrase $S[i, j]$ the strings $S[i, j + k], 0 \leq k < \tau$ are relevant substrings unless there is some longer relevant substring ending at position $j + k$. If $S[i', j']$ is a relevant substring then the string $S[j' + 1, n]$ is the *associated suffix*. There are at most $z\tau$ relevant substrings of S and equally many associated suffixes. The primary index is comprised by the following:

- A prefix search data structure T_D on the set of reversed relevant substrings.
- A prefix search data structure $\text{T}_{D'}$ on the set of associated suffixes.
- An orthogonal range reporting data structure R on the $z\tau \times z\tau$ grid. Consider a relevant substring $S[i, j]$. Let x denote the rank of $\text{rev}(S[i, j])$ in the lexicographical order of the reversed relevant substrings, let y denote the rank of its associated suffix $S[j + 1, n]$ in the lexicographical order of the associated suffixes. Then (x, y) is a point in R and along with it we store the pair (j, b), where b is the position of the rightmost phrase border contained in $S[i, j]$.

Note that every point (x, y) in R is induced by some relevant substring $S[i, j]$ and its associated suffix $S[j+1, n]$. If some prefix $P[1, k]$ is a suffix of $S[i, j]$ and the suffix $P[k+1, m]$ is a prefix of $S[j+1, n]$ then $S[j-k+1, j-k+m]$ is an occurrence of P and we can compute its exact location from k and j.

5.2 Searching

The data structure can be used to find the primary occurrences of a pattern P of length m when $m > \tau$. Consider the $O(m/\tau)$ prefix-suffix pairs $(P[1, i\tau], P[i\tau + 1, m])$ for $i = 1, \ldots, \lfloor m/\tau \rfloor$ and the pair $(P[1, m], \epsilon)$ in case m is not a multiple of τ. For each such pair, we do a prefix search for $\text{rev}(P[1, i\tau])$ and $P[i\tau + 1, m]$ in T_D and $\mathrm{T}_{D'}$, respectively. If either of these two searches report no matches, we move on to the next pair. Otherwise, let $[l, r]$, $[l', r']$ be the ranges reported from the search in T_D and $\mathrm{T}_{D'}$ respectively. Now we do a range reporting query on R for the rectangle $[l, r] \times [l', r']$. For each point reported, let (j, b) be the pair stored with the point. We report $j - i\tau + 1$ as the starting position of a primary occurrence of P in S.

Finally, in case m is not a multiple of τ, we need to also check the pair $(P[1, m], \epsilon)$. We search for $\text{rev}(P[1, m])$ in in T_D and ϵ in $\mathrm{T}_{D'}$. If the search for $\text{rev}(P[1, m])$ reports no match we stop. Otherwise, we do a range reporting query as before. For each point reported, let (j, b) be the pair stored with the point. To check that the occurrence has not been reported before we do as follows. Let k be the smallest positive integer such that $j - m + k\tau > b$. If $k\tau > m$ we report $j - m + 1$ as the starting position of a primary occurrence.

Correctness. We claim that the reported occurrences are exactly the primary occurrences of P. We first prove that all primary occurrences are reported correctly. Let $P = S[i', j']$ be a primary occurrence. As it is a primary occurrence, there must be some phrase $S[i^*, j^*]$ such that $i^* \leq i' \leq j^* \leq j'$. Let k be the smallest positive integer such that $i' + k\tau - 1 \geq j^*$. There are two cases: $k\tau \leq m$ and $k\tau > m$. If $k\tau \leq m$ then $P[1, k\tau]$ is a suffix of the relevant substring ending at $i' + k\tau - 1$. Such a relevant substring exists since $i' + k\tau - 1 < j^* + \tau$. Thus its reverse $\text{rev}(P[1, k\tau])$ prefixes a string s in D, while $P[k\tau + 1, m]$ is a prefix of the associated suffix $S[i' + k\tau, n] \in D'$. Therefore, the respective ranks of s and $S[i' + k\tau, n]$ in D and D' are plotted as a point in R which stores the pair $(i' + k\tau - 1, b)$. We will find this point when considering the prefix-suffix pair $(P[1, k\tau], P[k\tau + 1, m])$, and correctly report $(i' + k\tau - 1) - k\tau + 1 = i'$ as the starting position of a primary occurrence. If $k\tau > m$ then $P[1, m]$ is a suffix of the relevant substring ending in $i' + m - 1$. Such a relevant substring exists since $i' + m - 1 < i' + k\tau - 1 < j^* + \tau$. Thus its reverse prefixes a string in D and trivially ϵ is a prefix of the associated suffix. It follows as before that the ranks are plotted as a point in R storing the pair $(i' + m - 1, b)$ and that we find this point when considering the pair $(P[1, m], \epsilon)$. When considering $(P[1, m], \epsilon)$ we report $(i' + m - 1) - m + 1 = i'$ as the starting position of a primary occurrence if $k\tau > m$, and thus i' is correctly reported.

We now prove that all reported occurrences are in fact primary occurrences. Assume that we report $j - i\tau + 1$ for some i and j as the starting position of a primary occurrence in the first part of the procedure. Then there exist strings $\text{rev}(S[i', j])$ and $S[j + 1, n]$ in D and D' respectively such that $S[i', j]$ is suffixed by $P[1, i\tau]$ and $S[j + 1, n]$ is prefixed by $P[i\tau + 1, m]$. Therefore $j - i\tau + 1$ is the starting position of an occurrence of P. The string $S[i', j]$ is a relevant suffix and therefore there exists a border b in the interval $[j - \tau + 1, j]$. Since $i \geq 1$ the occurrence contains the border b and it is therefore a primary occurrence. If we report $j - m + 1$ for some j as the starting position of a primary occurrence in the second part of the procedure, then $\text{rev}(P[1, m])$ is a prefix of a string $\text{rev}(S[i', j])$ in D. It

follows immediately that $j - m + 1$ is the starting point of an occurrence. Since $m > \tau$ we have $j - m + 1 < j - \tau + 1$, and by the definition of relevant substring there is a border in the interval $[j - \tau + 1, j]$. Therefore the occurrence contains the border and is primary.

Complexity. We now consider the time complexity of the algorithm described. First we will argue that any primary occurrence is reported at most once and that the search finds at most two points in R identifying it. Let $S[i', j']$ be a primary occurrence reported when we considered the prefix-suffix pair $(P[1, k\tau], P[k\tau + 1, m])$ as in the proof of correctness. None of the pairs $(P[1, i\tau], P[i\tau + 1, m])$, where $i < k$ will identify this occurrence as $i' + i\tau - 1 < j$. None of the pairs $(P[1, h\tau], P[h\tau + 1, m])$, where $h > k$, will identify this occurrence. This is the case since $i' + h\tau - 1 > j + \tau - 1$, and from the definition of relevant substrings it follows that if $S[i, j]$ is a phrase, $S[a, b]$ is a relevant substring and $a < i$, then $b < i + \tau - 1$. Thus there are no relevant substrings that end after $j + \tau - 1$ and start before $i' < j$. Therefore, only one of the pairs $(P[1, i\tau], P[i\tau + 1, m])$ for $i = 1, \ldots, \lfloor m/x \rfloor$ identifies the occurrence. If $(k + 1)\tau > m$ then we might also find the occurrence when considering the pair $(P[1, m], \epsilon)$, but we do not report i' as $k\tau \leq m$.

After preprocessing P in $O(m)$ time, we can do the $O(m/\tau)$ prefix searches in total time $O(m + m/\tau(m/x + \lg x))$ where x is a positive integer by Lemma 3. Using the range reporting data structure by Chan et al. [6] each range reporting query takes $(1 + k) \cdot O(B \lg \lg(z\tau))$ time where $2 \leq B \leq \lg^{\epsilon}(z\tau)$ and k is the number of points reported. As each such point in one range reporting query corresponds to the identification of a unique primary occurrence of P, which happens at most twice for every occurrence we charge $O(kB \lg \lg(z\tau))$ to reporting the occurrences. The total time to find all primary occurrences is thus $O(m + \frac{m}{\tau}(\frac{m}{x} + \lg x + B \lg \lg(z\tau)) + \text{occ}\, B \lg \lg(z\tau))$ where occ is the number of primary and secondary occurrences of P.

5.3 Prefix Search Verification

The prefix data structure from Lemma 3 gives no guarantees of correct answers when the query pattern does not prefix any of the indexed strings. If the prefix search gives false-positives, we may end up reporting occurrences of P that are not actually there. We show how to solve this problem after introducing a series of tools that we will need.

Straight line programs. A *straight line program* (SLP) for a string S is a context-free grammar generating the single string S.

▶ **Lemma 4** (Rytter [36], Charikar et al. [7]). *Given an LZ77 parse Z of length z producing a string S of length n we can construct a SLP for S of size $O(z \lg(n/z))$ in time $O(z \lg(n/z))$.*

The construction from Rytter [36] produces a balanced grammar for every consecutive substring of length n/z of S after a preprocessing step transforms Z such that no compression element is longer than n/z. The height of this balanced grammar is $O(\lg n)$ and this immediately yields extracting of any substring $S[i, j]$ in time $O(\lg(n) + j - i)$. We give a simple solution to reduce this to $O(\lg(n/z) + j - i)$, that also supports computation of the fingerprint of a substring in $O(\lg(n/z))$ time.

▶ **Lemma 5.** *Given an LZ77 parse Z of length z producing a string S of length n we can build a data structure that for any substring $S[i, j]$ can extract $S[i, j]$ in $O(\lg(n/z) + j - i)$ time and compute the fingerprint $\phi(S[i, j])$ in $O(\lg(n/z))$ time. The data structure uses $O(z \lg(n/z))$ space and $O(n)$ construction time.*

Proof. Assume for simplicity that n is a multiple of z. We construct the SLP producing S from Z. Along with every non-terminal of the SLP we store the size and fingerprint of its expansion. Let $s_1, s_2, \ldots s_z$ be consecutive length n/z substrings of S. We store the balanced grammar producing s_i along with the fingerprint $\phi(S[1,(i-1)n/z])$ at index i in a table A.

Now we can extract s_i in $O(n/z)$ time and any substring $s_i[j,k]$ in time $O(\lg(n/z)+k-j)$. Also, we can compute the fingerprint $\phi(s_i[j,k])$ in $O(\lg(n/z))$ time. We can easily do a constant time mapping from a position in S to the grammar in A producing the substring covering that position and the corresponding position inside the substring. But then any fingerprint $\phi(S[1,j])$ can be computed in time $O(\lg(n/z))$. Now consider a substring $S[i,j]$ that starts in s_k and ends in $s_l, k < l$. We extract $S[i,j]$ in $O(\lg(n/z)+j-i)$ time by extracting the appropriate suffix of s_k, all of s_m for $k < m < l$ and the appropriate prefix of s_l. Each of the fingerprints stored by the data structure can be computed in $O(1)$ time after preprocessing S in $O(n)$ time. Thus table A is filled in $O(z)$ time and by Lemma 4 the SLPs stored in A uses a total of $O(z\lg(n/z))$ space and construction time. ◀

Verification of fingerprints. We need the following lemma for the verification.

▶ **Lemma 6** (Bille et al. [5]). *Given a string S of length n, we can find a fingerprinting function ϕ that is collision-free for all length l substrings of S where l is a power of two in $O(n\lg n)$ expected time.*

5.3.1 Verification Technique

Our verification technique is identical to the one given by Gagie et al. [18] and involves a simple modification of the search for long primary occurrences. By using Lemma 5 instead of bookmarking [18] for extraction and fingerprinting and because we only need to verify $O(m/\tau)$ strings, the verification procedure takes $O(m + m/\tau\lg(n/z))$ time and uses $O(z\lg(n/z))$ space. See Appendix A.1 for details.

6 Short Primary Occurrences

We now describe a simple data structure that can find primary occurrences of P in time $O(m+\mathrm{occ})$ using space $O(z\tau)$ whenever $m \leq \tau$ where τ is a positive integer.

Let Z be the LZ77 parse of the string S of length n. Let $Z[i] = S[s_i, e_i]$ and define F to be the union of the strings $S[k, \min\{e_i+\tau, n\}]$ where $\max\{1, s_i, e_i-\tau\} \leq k \leq e_i$ for $i = 1, 2, \ldots z$. There are at most $z\tau$ such strings, each of length $O(\tau)$ and they are all suffixes of the z length 2τ substrings of S starting τ positions before each border position. We store these substrings along with the compact trie T_F over the strings in F. The edge labels of T_F are compactly represented by storing references into one of the substrings. Every leaf stores the starting position in S of the string it represents and the position of the leftmost border it contains.

The combined size of T_F and the substrings we store is $O(z\tau)$ and we simply search for P by navigating vertices using perfect hashing [16] and matching edge labels character by character. Now either $\mathrm{locus}(P) = \perp$ in which case there are no primary occurrences of P in S; otherwise, $\mathrm{locus}(P) = v$ for some vertex $v \in \mathrm{T}_F$ and thus every leaf in the subtree of v represents a substring of S that is prefixed by P. By using the indices stored with the leaves, we can determine the starting position for each occurrence and if it is primary or secondary. Because each of the strings in F start at different positions in S, we will only find an occurrence once. Also, it is easy to see that we will find all primary occurrences because

of how the strings in F are chosen. It follows that the time complexity is $O(m + \text{occ})$ where occ is the number of primary and secondary occurrences.

7 The Secondary Index

Let Z be the LZ77 parse of length z representing the string S of length n. We find the secondary occurrences by applying the most recent range reporting data structure by Chan et al. [6] to the technique described by Kärkkäinen and Ukkonen [25]. This gives us a secondary index using $O(z \lg \lg z)$ space and $O(\text{occ} \lg \lg n)$ time for reporting all secondary occurrences. For details see Appendix A.2.

8 The Compressed Index

We obtain our final index by combining the primary index, the verification data structure and the secondary index. We use the transformed LZ77 parse generated by Lemma 4 when building our primary index. Therefore no phrase will be longer than n/z and therefore any primary occurrence of P will have a prefix $P[1, k]$ where $k \leq n/z$ that is a suffix of some phrase. It then follows that we need only consider the multiples $(P[1, i\tau], P[i\tau + 1, m])$ for $i < \lfloor \frac{n/z}{\tau} \rfloor$ when searching for long primary occurrences. This yields the following complexities:

- $O(m + \frac{\min\{m,n/z\}}{\tau}(\frac{m}{x} + \lg x + B \lg \lg(z\tau)) + \text{occ}\ B \lg \lg(z\tau))$ time and $O(z\tau \lg_B \lg(z\tau))$ space for the index finding long primary occurrences where x and τ are positive integers and $2 \leq B \leq \lg^\epsilon(z\tau)$.
- $O(m + \text{occ})$ time and $O(z \lg(n/z))$ space for the index finding short primary occurrences.
- $O(m + m/\tau \lg(n/z))$ time and $O(z \lg(n/z))$ space for the verification data structure.
- $O(\text{occ} \lg \lg n)$ time and $O(z \lg \lg z)$ space for the secondary index.

If we fix x at n/z we have $\frac{\min\{m,n/z\}}{\tau} \frac{m}{x} \leq m$ in which case we obtain the following trade-off simply by combining the above complexities.

▶ **Theorem 7.** *Given a string S of length n from an alphabet of size σ compressed using LZ77 to a string of length z we can build a compressed-index supporting substring queries in $O(m + \frac{m}{\tau}(\lg(n/z) + B \lg \lg(z\tau)) + \text{occ}(B \lg \lg(z\tau) + \lg \lg n))$ time using $O(z(\lg(n/z) + \tau \lg_B \lg(z\tau) + \lg \lg z))$ space for any query pattern P of length m where $2 \leq B \leq \lg^\epsilon(z\tau)$, $0 < \epsilon < 1$ and τ is a positive integer.*

We note that none of our data structures assume constant sized alphabet and thus Theorem 7 holds for any alphabet size.

Due to lack of space the description and analysis of the preprocessing have been moved to Appendix 8.2.

8.1 Trade-offs

Theorem 7 gives rise to a series of interesting time-space trade-offs.

▶ **Corollary 8.** *Given a string S of length n from an alphabet of size σ compressed using LZ77 into a string of length z we can build a compressed-index supporting substring queries in*

(i) $O(m(1 + \frac{\lg \lg z}{\lg(n/z)}) + \text{occ} \lg \lg n)$ *time using* $O(z \lg(n/z) \lg \lg z)$ *space, or*
(ii) $O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \text{occ}(\lg \lg n + \lg^\epsilon z))$ *time using* $O(z \lg(n/z))$ *space, or*
(iii) $O(m \lg^\epsilon(n/z) + \text{occ} \lg \lg n)$ *time using* $O(z \lg(n/z))$ *space, or*
(iv) $O(m + \text{occ} \lg \lg n)$ *time using* $O(z(\lg(n/z) \lg \lg z + \lg \lg^2 z))$ *space, or*

(v) $O(m + \mathrm{occ}(\lg\lg n + \lg^{\epsilon} z))$ *time using* $O(z(\lg(n/z) + \lg^{\epsilon'} z))$ *space.*
for any $0 < \epsilon < 1$ *and* $0 < \epsilon' < 1$.

Proof. For (i) set $B = 2$ and $\tau = \lg(n/z)$, for (ii) set $B = \lg^{\epsilon} z$ and $\tau = \lg(n/z)$, for (iii) set $B = 2$ and $\tau = \lg^{\epsilon'} n/z$ for some $0 < \epsilon' < 1$, for (iv) set $B = 2$ and $\tau = \lg(n/z) + \lg\lg z$, for (v) set $B = \lg^{\epsilon'}(z)$ and $\tau = \lg(n/z) + \lg^{\epsilon} z$. ◀

The leading term in the time complexity of Corollary 8 (i) is $O(m)$ whenever $\lg\lg(z) = O(\lg(n/z))$ which is true when $z = O(n/\lg n)$, i.e. for all strings that are compressible by at least a logarithmic fraction. For $\sigma = O(1)$ we have $z = O(n/\lg n)$ all strings [34] and thus Theorem 1 (i) follows immediately. Corollary 8 (ii) matches previous best space bounds but obtains a leading term of $O(m)$ for any polynomial compression rate. Theorem 1 (ii) is a weaker version of this because it assumes constant sized alphabet and therefore follows immediately. Corollary 8 (iii) matches the space and time for reporting occurrences of previous best bounds by Gagie et al. [18] but with a leading term of $O(m\lg^{\epsilon}(n/z))$ compared to a leading term of $O(m\lg m)$. Corollary 8 (iv) and (v) show how to guarantee the fast query times with leading term $O(m)$ without the assumptions on compression ratio that (i) and (ii) require to match this, but at the cost of increased space.

8.2 Preprocessing

We now consider the preprocessing time of the data structure. Let Z be the LZ77 parse of the string S of length n let T_D and $\mathrm{T}_{D'}$ be the compact tries used in the index for long primary occurrences. The compact trie T_D index $O(z\tau)$ substrings of S with overall length $O(n\tau)$. Thus we can construct the trie in $O(n\tau)$ time by sorting the strings and successively inserting them in their sorted order [1]. The compact tries $\mathrm{T}_{D'}$ index $z\tau < n$ suffixes of S and can be built in $O(n)$ time using $O(n)$ space [10]. The index for short primary occurrences is a generalized suffix tree over z strings of length $O(\tau)$ with total length $z\tau < n$ and is therefore also built in $O(n)$ time. The dictionaries used by the prefix search data structures and for trie navigation contain $O(z\tau)$ keys and are built in expected linear time using perfect hashing [16]. The range reporting data structures used by the primary and secondary index over $O(z\tau)$ points are built in $O(z\tau\lg(z\tau))$ expected time using Lemma 9.

Building the SLP for our verification data structure takes $O(z\lg(n/z))$ time using Lemma 4 and finding an appropriate fingerprinting function ϕ takes $O(n\lg n)$ expected time using Lemma 6. The prefix search data structures T_D and $\mathrm{T}_{D'}$ also require that ϕ is collision-free for the x-prefixes, fat prefixes and the prefixes with pseudo fat lengths. There are at most $O(z\tau\lg n)$ such prefixes [2]. If we compute these fingerprints incrementally while doing a traversal of the tries, we expect all the fingerprints to be unique. We simply check this by sorting the fingerprints in linear time and checking for duplicates by doing a linear scan. If we choose a prime $p = \Theta(n^5)$ for the fingerprinting function then the probability of a collision between any two strings is $O(1/n^4)$ [35] and by a union bound over the $O((n\lg n)^2)$ possible collisions the probability that ϕ is collision-free is at least $1 - 1/n$. Thus the expected time to find our required fingerprinting function is $O(n + n\lg n)$.

All in all, the preprocessing time for our combined index is therefore expected $O(n\lg n + n\tau)$.

References

1. Arne Andersson and Stefan Nilsson. A new efficient radix sort. In Shafi Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, pages 714–721. IEEE Computer Society, 1994. doi:10.1109/SFCS.1994.365721.
2. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In Mark de Berg and Ulrich Meyer, editors, *Proceedings of the 18th Annual European Symposium on Algorithms (ESA 2010)*, volume 6346 of *LNCS*, pages 427–438. Springer, 2010. doi:10.1007/978-3-642-15775-2_37.
3. Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, volume 9133 of *LNCS*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.
4. Djamal Belazzougui, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2015 Data Compression Conference (DCC 2015)*, pages 83–92. IEEE, 2015. doi:10.1109/DCC.2015.69.
5. Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. In Juha Kärkkäinen and Jens Stoye, editors, *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM 2012)*, volume 7354 of *LNCS*. Springer, 2012. doi:10.1007/978-3-642-31265-6_24.
6. Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry (SocG 2011)*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.
7. Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.
8. Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016. doi:10.1016/j.is.2016.04.002.
9. Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *LNCS*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19.
10. Martin Farach. Optimal suffix tree construction with large alphabets. In Anna Karlin, editor, *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143, Washington, DC, USA, 1997. IEEE Computer Society. doi:10.1109/SFCS.1997.646102.
11. Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998. doi:10.1007/PL00009202.
12. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In Avrim Blum, editor, *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.
13. Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In S. Rao Kosaraju, editor, *Proceedings of the 12th Annual ACM-SIAM Symposium on*

Discrete Algorithms (SODA 2001), pages 269–278. ACM/SIAM, 2001. URL: http://dl.acm.org/citation.cfm?id=365411.365458.

14 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

15 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007. doi:10.1145/1240233.1240243.

16 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

17 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012. doi:10.1007/978-3-642-28332-1_21.

18 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In Alberto Pardo and Alfredo Viola, editors, *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN 2014)*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014. doi:10.1007/978-3-642-54423-1_63.

19 Travis Gagie and Simon J. Puglisi. Searching and indexing genomic databases via kernelization. *Front. Bioeng. Biotechnol.*, 3:12, 2015. doi:10.3389/FBIOE.2015.00012.

20 Jean-Loup Gailly and Mark Adler. GNU zip, 1992. URL: http://www.gzip.org/.

21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In Martin Farach-Colton, editor, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

22 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In J. Ian Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 636–645. SIAM, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982888.

23 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406. ACM, 2000. doi:10.1145/335305.335351.

24 Juha Kärkkäinen and Erkki Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998. doi:10.1007/PL00009205.

25 Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In Nivio Ziviani, Ricardo Baeza-Yates, and Katia Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing (WSP 1996)*, pages 141–155. Carleton University Press, 1996.

26 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

27 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

28 Moshe Lewenstein. Orthogonal range searching for text indexing. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *LNCS*, pages 267–302. Springer, 2013. doi:10.1007/978-3-642-40273-9_18.

29 Veli Mäkinen. Compact suffix array. In Raffaele Giancarlo and David Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 305–319. Springer, 2000. doi:10.1007/3-540-45123-4_26.

30 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.

31 Donald R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968. doi:10.1145/321479.321481.

32 Gonzalo Navarro. Indexing highly repetitive collections. In S. Arumugam and W. F. Smyth, editors, *Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA 2012)*, volume 7643 of *LNCS*, pages 274–279. Springer, 2012. doi:10.1007/978-3-642-35926-2_29.

33 Gonzalo Navarro. *Compact Data Structures: A practical approach*. Cambridge University Press, 2016. doi:10.1017/CBO9781316588284.

34 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), April 2007. doi:10.1145/1216370.1216372.

35 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In Daniel A. Spielman, editor, *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009)*, pages 315–323. IEEE Computer Society, 2009. doi:10.1109/FOCS.2009.11.

36 Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

37 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

A Appendix

A.1 Verification Technique

Consider the string S of length n that we wish to index and let Z be the $LZ77$ parse of S. The verification data structure is given by Lemma 5. Consider the prefix search data structure $\mathrm{T}_{D'}$ as given in Section 5.1 and let ϕ be the fingerprinting function used by the prefix search, the case for T_D is symmetric. We alter the search for primary occurrences such that it first does the $O(m/\tau)$ prefix searches, then verifies the results and discards false-positives before moving on to do the $O(m/\tau)$ range reporting queries on the verified results. We also modify ϕ using Lemma 6 to be collision-free for all substrings of the indexed strings which length is a power of two.

Let $Q_1, Q_2, \ldots Q_j$ be the all the suffixes of P for which the prefix search found a locus candidate, let the candidates be $v_1, v_2, \ldots v_j \in \mathrm{T}_{D'}$ and let p_i be $\mathrm{str}(v_i)[1, |Q_i|]$. Assume that $|Q_i| < |Q_{i+1}|$, and let 2-suf(Q) and 2-pre(Q) denote the fingerprints using ϕ of the suffix and prefix respectively of length $2^{\lfloor \lg |Q| \rfloor}$ of some string Q. The verification progresses in iterations. Initially, let $a = 1$, $b = 2$ and for each iteration do as follows:

1. 2-suf$(Q_a) \neq$ 2-suf(p_a) or 2-pre$(Q_a) \neq$ 2-pre(p_a): Discard v_a and set $a = a + 1$ and $b = b + 1$.
2. 2-suf$(Q_a) =$ 2-suf(p_a) and 2-pre$(Q_a) =$ 2-pre(p_a), let $R = p_b[|p_a| - |p_b| + 1, |p_a|]$.
   a. 2-suf$(R) =$ 2-suf(Q_a) and 2-pre$(R) =$ 2-pre(Q_a): set $a = a + 1$ and $b = b + 1$.
   b. 2-suf$(R) \neq$ 2-suf(Q_a) or 2-pre$(R) \neq$ 2-pre(Q_a): discard v_b and set $b = b + 1$.

3. $b = j + 1$: If all vertices have been discarded, report no matches. Otherwise, let v_f be the last vertex considered, that was not discarded. Compare p_f to Q_f and if equal, report all non-discarded vertices as verified. Otherwise discard all vertices and report no matches.

Consider the correctness and complexity of the algorithm. In case 1, clearly, p_a does not match Q_a and thus v_a must be a false-positive. Now observe that because Q_i is a suffix of P, it is also a suffix of $Q_{i'}$ for any $i < i'$. Thus in case 2 (b), if R does not match Q_a then v_b must be a false-positive. In case 2 (a), both v_a and v_b may still be false-positives, yet by Lemma 6, p_a is a suffix of p_b because 2-suf(p_a) = 2-suf(R) and 2-pre(p_a) = 2-pre(R). Finally, in case 3, v_f is a true positive if and only if $p_f = Q_f$. But any other non-discarded vertex $v_i \neq v_f$ is also only a true positive if $p_f = Q_f$ because p_i is a suffix of p_f and Q_i is a suffix of Q_p.

The algorithm does j iterations and fingerprints of substrings of P can be computed in constant time after $O(m)$ preprocessing. Every vertex $v \in \mathrm{T}_{D'}$ represents one or more substrings of S. If we store the starting index in S of one of these substrings in v when constructing $\mathrm{T}_{D'}$ we can compute the fingerprint of any substring $\mathrm{str}(v)[i, j]$ by computing the fingerprint of $S[i' + i - 1, i' + j - 1]$ where i' is the starting index of one of the substring of S that v represents. By Lemma 5, the fingerprint computations take $O(\lg(n/z))$ time and because $j \leq m/\tau$ the total time complexity of the algorithm is $O(m + m/\tau \lg(n/z))$.

A.2 Secondary Index

Let Z be the LZ77 parse of length z representing the string S of length n. We find the secondary occurrences by applying the most recent range reporting data structure by Chan et al. [6] to the technique described by Kärkkäinen and Ukkonen [25] which is inspired by the ideas of Farach and Thorup [11].

Let $X \subseteq \{0, \ldots, u\}^d$ be a set of points in a d-dimensional grid. The *orthogonal range reporting problem* in d-dimensions is to compactly represent X while supporting *range reporting queries*, that is, given a rectangle $R = [a_1, b_2] \times \cdots \times [a_d, b_d]$ report all points in the set $R \cap X$. We use the following results for 2-dimensional range reporting:

▶ **Lemma 9** (Chan et al. [6]). *For any set of n points in $[0, u] \times [0, u]$ and $2 \leq B \leq \lg^\epsilon n, 0 < \epsilon < 1$ we can solve 2-d orthogonal range reporting with $O(n \lg n)$ expected preprocessing time, $O(n \lg_B \lg n)$ space and $(1 + k) \cdot O(B \lg \lg u)$ query time where k is the number of occurrences inside the rectangle.*

Let $o_1, \ldots o_{\mathrm{occ}}$ be the starting positions of the occurrences of P in S ordered increasingly. Assume that o_h is a secondary occurrence such that $P = S[o_h, o_h + m - 1]$. Then by definition, $S[o_h, o_h + m - 1]$ is a substring the prefix $S[i, j - 1]$ of some phrase $S[i, j]$ and there must be an occurrence of P in the source of that phrase. More precise, let $S[k, l] = S[i, j - 1]$ be the source of the phrase $S[i, j]$ then $o_{h'} = k + o_h - i$ is an occurrence of P for some $h' < h$. We say that $o_{h'}$, which may be primary or secondary, is the source occurrence of the secondary occurrence o_h given the LZ77 parse of S. Thus every secondary occurrence has a source occurrence. Note that it follows from the definition that no primary occurrence has a source occurrence.

We find the secondary occurrences as follows: Build a range reporting data structure Q on the $n \times n$ grid and if $S[i, j]$ is a phrase with source $S[i', j']$ we plot a point (i', j') and along with it we store the phrase start i.

Now for each primary occurrence o found by the primary index, we query Q for the rectangle $[0, o] \times [o + m - 1, n]$. The points returned are exactly the occurrences having

o as source. For each point (x, y) and phrase start i reported, we report an occurrence $o' = i + o - x$ and recurse on o' to find all the occurrences having o' as source.

Because no primary occurrence have a source, while all secondary occurrences have a source, we will find exactly the secondary occurrences.

The range reporting structure Q is built using Lemma 9 with $B = 2$ and uses space $O(z \lg \lg z)$. Exactly one range reporting query is done for each primary and secondary occurrence each taking $O((1 + k) \lg \lg n)$ where k is the number of points reported. Each reported point identifies a secondary occurrence, so the total time is $O(\text{occ} \lg \lg n)$.