Engineering Zuffix Arrays

Paolo Boldi
Università degli Studi di Milano, Italy

Stefano Marchini
Università degli Studi di Milano, Italy

Sebastiano Vigna
Università degli Studi di Milano, Italy

Abstract

Searching patterns in long strings is a classical algorithmic problem with countless practical applications. Suffix trees and suffix arrays (and their variants) are a long-established solution that yields linear-time search (in the size of the pattern). In [5] it is shown that a z-map gadget can be attached to (enhanced) suffix arrays to improve their theoretical query time, obtaining a data structure called *zuffix array*. The main contribution of this paper is to show that a carefully engineered implementation of the z-map gadget does provide significant speedups with respect to enhanced suffix arrays on real-world datasets, albeit doubling the required space. In particular, for large alphabets we observe a sevenfold improvement in query time with respect to enhanced suffix arrays; even in the worst case (small alphabets), the query time is almost halved. Thus, zuffix arrays provide a very interesting new point in the space-time tradeoff spectrum.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Suffix trees, suffix arrays, z-fast tries

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.2

Supplementary Material *Software (Source Code)*: `https://github.com/smarchini/zuffix/tree/reproduce`

Funding This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.

1 Introduction

The suffix array [19] of a string s over an ordered alphabet Σ of σ characters is the array of the starting points of the suffixes of $s\$$ (where \$ is a character larger than any element of Σ), sorted lexicographically by the corresponding suffix. Suffix arrays are an extremely effective way of looking for all the occurrences of a pattern in a string, as once they are built (with some additional ancillary data) search requires only an amount of work linear in the length of the search pattern. A large body of research has gone into building and representing suffix arrays efficiently (e.g., in compressed form) [14, 8, 20, 18, 16, 22, 23, 10, 21, 6].

Interestingly, at the price of two additional (and very compressible) arrays of integers a suffix array can be used to represent (and navigate) implicitly the suffix tree associated with the string s [1]. Suffix trees are a special type of trie [17], which naturally suggests the possibility to enhance searches by using a z-map [2], obtaining what was called a *zuffix tree* in [5]. With this extra gadget in place, given a pattern p of length m, after a preprocessing requiring time $O(1 + (m \log \sigma)/w)$ (w being the machine word size), searches can be performed in $O(\log m)$ search steps, each accessing a constant amount of information. The result is obtained only with high probability, and requires a verification phase (accessing m

characters), but the interesting fact is that the characters are accessed in sequential fashion at less than σ different positions of the original text s. In other words, the described technique makes it possible to find with high probability the occurrences of pattern p in s using time $O((m \log \sigma)/w + \log m + \sigma)$ and $O((m \log \sigma)/B + \log m + \sigma)$ I/Os in the cache-oblivious model.

Whether (and under which conditions) this theoretical advantage in the cache-oblivious model reflects an improvement in actual searches is the question we aim to answer in this paper. We start from an enhanced suffix array, and we add a carefully engineered implementation of the z-map gadget; our experiments show that the resulting data structure does provide significant speedups with respect to enhanced suffix arrays on real-world datasets, albeit doubling the required space. It is worth noting that a z-map can be attached to *any* trie-based data structure, so this paper should be seen as a first effort in the direction of improving different types of structures based on suffix trees and suffix arrays.

All code used in our experiments, with full reproducibility instructions, is available at `https://github.com/smarchini/zuffix/tree/reproduce`.

2 Notation and Tools

2.1 Notation

Let Σ be a fixed alphabet (of cardinality σ) not including the special symbol \$, and define $\hat{\Sigma} = \Sigma \cup \{\$\}$. The alphabet Σ comes endowed with a specified (arbitrary) total order, that is inherited by $\hat{\Sigma}$ with the proviso that \$ is larger than any other character. We use $\leq$ to denote the induced lexicographic order on $\hat{\Sigma}^*$, whereas $\preceq$ is used to denote the prefix order.

If $x \in \hat{\Sigma}^*$ is a string, x juxtaposed with an interval is the substring of x with those indices (indices start from 0). Thus, for instance, $x[a\,..\,b]$ is the substring of x starting at position a (inclusive) and ending at position b (inclusive). We will write $x[a]$ for $x[a\,..\,a]$ and $x[a\,..\,]$ for $x[a\,..\,|x|-1]$. By definition, $x[|x|\,..\,] = \varepsilon$.

2.2 Machine model

We analyze our algorithms on a unit-cost word RAM with word size w in the cache-oblivious model [11]. In this model, the machine has a two-level memory hierarchy, where the fast level has an unknown size of M words and the slow level has an unbounded size and is where our data reside. We assume that the fast level plays the role of a cache for the slow level with an optimal replacement strategy where the transfers (a.k.a. I/Os) between the two levels are done in blocks of unknown size of $B \leq M$ words; the I/O cost of an algorithm is the total number of such block transfers. *Scanning* is a fundamental building block in the design of cache-oblivious algorithms: given an array of N contiguous items the I/Os required for scanning it completely is $O(1 + N/B)$.

2.3 Tries

A *(compacted) trie* [17] over Σ is a rooted tree such that

- every node α is endowed with a string $c_\alpha \in \Sigma^*$ (the *compacted path* of α);
- every arc connecting an internal node α with one of its children α' is labelled with a character $c_{\alpha,\alpha'} \in \Sigma$ and $c_{\alpha,\alpha'} \neq c_{\alpha,\alpha''}$ for any two distinct children α' and α'' of α;
- every internal node has at least two children.

For every node α of a compacted trie, we define its *name* $n_\alpha \in \Sigma^*$ and its *extent* $e_\alpha \in \Sigma^*$ recursively as follows:

- $n_{\text{root}} = \varepsilon$;
- $e_\alpha = n_\alpha c_\alpha$;
- if α' is a child of α, then $n_{\alpha'} = e_\alpha c_{\alpha,\alpha'}$.

For any given finite nonempty prefix-free language $L \subseteq \Sigma^*$, the *compacted trie of* L is the only compacted trie $T(L)$ over Σ such that L is the set of all the extents of the leaves of $T(L)$.

In Figure 1, we show an example of a binary trie with the nomenclature just introduced (and some more that will be introduced below).

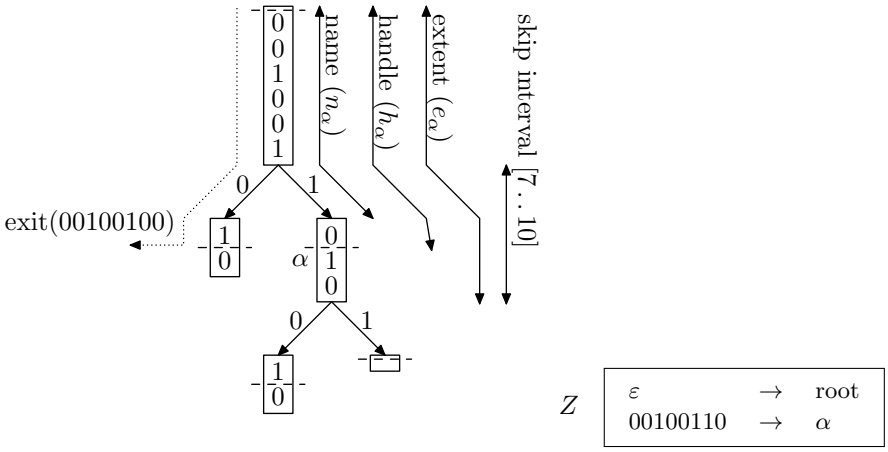


Figure 1 (above) The compacted trie $T(L)$ (with the corresponding nomenclature) and its z-map. Here $\Sigma = \{0, 1\}$ and $L = \{001001010, 0010011010010, 00100110101\}$.

Given a compacted trie over Σ, and given a string p, we let $\text{exit}(p)$ (the *exit node of* p) be the node having a name that is the longest possible prefix of p. (See Figure 1 for an example of an exit node).

Note that:

▶ **Proposition 1** ([5]). *Let $L \subseteq \Sigma^*$ be finite, nonempty, and prefix-free, and consider the trie $T(L)$. A string $p \in \Sigma^*$ is a prefix of some element of L if and only if $p \preceq e_{\text{exit}(p)}$. Moreover, if the latter happens then the set*

$$\{e_\alpha \mid \alpha \text{ is a leaf descendant of } \text{exit}(p)\}$$

is precisely the set of all $x \in L$ such that $p \preceq x$.

In the following, we shall always assume that Σ is a fixed alphabet, $\sigma = |\Sigma|$ and that $L \subseteq \Sigma^*$ is finite, nonempty and prefix-free.

3 Z-fast tries

Let us assume that we have built the trie $T(L)$ for a given language L of size n. Proposition 1 gives an easy way to determine if a string p of length m is a prefix of some element of L: it is enough to locate the node $\gamma = \text{exit}(p)$ and then to check whether p is a prefix of the extent of γ or not. Moreover, the second part of the statement suggests which elements of L have p as a prefix.

Locating exit(p) can be done trivially in $O(m\sigma)$ steps, going down in the trie starting from the root. An alternative faster solution is that of enriching the compacted trie with a kind of acceleration map, the *z-map* [2], which makes it possible navigate the trie using a number of accesses to the map logarithmic in the length of the search string.

3.1 Short strings

The idea comes in two flavors, depending on the length of the strings in the set L. Let us start with the (simplest) case of "short" strings. The *2-fattest number* of an interval $[a\,..\,b]$ of non-negative integers is the (unique) integer in $[a\,..\,b]$ that is divisible by the largest power of two. The *handle* h_α of a node α of a trie is the prefix of e_α whose length is the 2-fattest number in $[|n_\alpha|\,..\,|e_\alpha|]$ (the *skip interval* of α).

In Figure 1 we show the (length of the) handles of each node, including the leaves: the handle is the string starting from the root and ending just above the dashed lines you can see in each node.

▶ **Definition 2** (z-map). *The* z-map $Z(-)$ *for the trie* $T(L)$ *is a map from elements of* Σ^* *to nodes in the trie, which maps* h_α *to* α *for each* internal *node* α*.*

The z-map can be stored using any static dictionary with constant-time access; we assume that the dictionary returns the special value $\perp$ whenever it is queried with a key that is not in the dictionary.

The usefulness of the z-map is made evident by the following result:

▶ **Theorem 3** ([5]). *If the length of the strings in* L *is bounded by* $O(w/\log\sigma)$*, using the z-map for the trie* $T(L)$ *it is possible to establish if a pattern* $p \in \Sigma^*$ *of length* m *is a prefix of some string of* L *in* $O(\log m + \sigma)$ *time and I/Os.*

The algorithm, presented in [5], performs a sort of a binary search (called *fat binary search*), with at most one access to the actual trie structure (to enumerate the children of a node). Moreover, the comparisons between p and other strings require at most σ scans of overall m characters.

3.2 Long strings

If the length of the strings in L is not bounded by $O(w/\log\sigma)$, the map $Z(-)$ described above uses superlinear space and superconstant time at every access. This is why the "long string" version of dynamic z-fast tries [3] replaces handles with *signatures*: in the z-map, instead of storing pairs (h_α, α) we store pairs $(H(h_\alpha), \alpha)$ where $H(-)$ is a suitably chosen signature hash function. The signature-based version is designed to work with sets of at most $2^{O(w)}$ strings of length up to $2^{O(w)}$, but this time the fat binary search returns the correct result only with high probability.

If we use signatures of size $(c+\varepsilon)\log n$, with $c \geq 2$, we will find distinct hash values for all handles after a constant expected number of attempts. Indeed, under a full randomness assumption, the probability of having a hash collision between distinct handles is at most

$$1 - e^{-n^2/2^{1+(c+\varepsilon)\log n}} = 1 - e^{-n^2/2n^{c+\varepsilon}} \leq \frac{1}{2n^{c-2+\varepsilon}} \to 0$$

as $n \to \infty$. Once we are sure that the signatures are all distinct, in estimating the probability of error of a fat binary search we have to care just about at most $\log m$ false-positive results in queries performed during the fat binary search, which by the union bound happens with probability at most

$$2^{-(c+\varepsilon)\log n}\log m = \frac{1}{n^c n^\varepsilon}O(w) = o\left(\frac{1}{n^c}\right).$$

Note that each time we have to query the z-map, we must compute the hash of a potentially long prefix: for this purpose, the dynamic z-fast trie uses hash functions that can hash any prefix of the pattern p in constant time after preprocessing p in time $O(1 + (m \log \sigma)/w)$ and storing a linear amount of information (see Section 5).

Thus, for the signature-based case, we have:

▶ **Theorem 4** ([5]). *Under the full randomness assumption, let $c \geq 2$ and assume that $Z(-)$ stores $((c+\varepsilon) \log n)$-bit hash values without collisions. Then, in time $O((m \log \sigma)/w + \log m + \sigma)$ and with $O((m \log \sigma)/B + \log m + \sigma)$ I/Os it is possible to establish if a pattern $p \in \Sigma^*$ of length m is prefix of some string of L; the result will be correct with probability at least $1 - o(1/n^c)$; when the algorithm detects that an error has occurred, the correct result is obtained by resorting to the standard naive search on the trie, which requires $O(m\sigma)$ time and I/Os.*

4 Suffix Trees, Suffix Arrays, and Zuffification

Given a string $s \in \Sigma^*$, the *suffix tree* [24] $T(s)$ is the trie over the alphabet $\hat{\Sigma}$ containing all suffixes of $s\$$. Although one can build in linear time the suffix tree of a string, there are approaches based on *suffix arrays* [19] that are much more efficient in practice. The suffix array sa of s is the permutation of $\{0, 1, \ldots, n\}$ such that

$$s\$[\mathrm{sa}[0]\,..\,] < s\$[\mathrm{sa}[1]\,..\,] < \cdots < s\$[\mathrm{sa}[n]\,..\,].$$

Not only there are linear-time algorithms to build the suffix array: at the price two additional (and very compressible) arrays of integers, the suffix array can be made into a so-called *enhanced suffix array*, and then used to represent and navigate implicitly the suffix tree [1].

The z-map gadget described in Section 3 is applicable to all prefix-free languages, so in particular it can be used for suffix trees, and since the enhanced suffix array is equivalent to the suffix tree, it can be used to build the z-map over the suffix tree/array: this operation is called *zuffification* of the suffix tree/array [5].

To build the z-map, we perform a depth-first visit of the suffix tree: in fact, we will simulate it on a suffix array. For each internal node, we can compute its handle (as explained above), the corresponding hash, and store in a static dictionary the correspondence between the hash and the node; hash collisions will cause some correspondence to be wrong, but we will handle this problem as part of the resilience of fat binary search to hash collisions. Computing a hash for a string x needs time $O(1+(|x| \log \sigma)/w)$, which would imply quadratic construction and preprocessing time: in the next sections, we are going to discuss how to reduce it to linear time.

5 Constant Time Prefix Hashing

We start by studying how to compute the hash of a prefix of a string s in constant time. Note that in theory this is not enough to obtain linear construction time, but we will see that in practice this kind of approach works very well.

When looking for the exit node of a string we query the z-map on prefixes of the pattern. We will use a scheme that scans the entire string only once and enables computing the signature of any prefix in constant time. This scheme exploits the *pause&resume* technique, exposed by several general-purpose hash functions: to compute the signature of large files or streaming data it is common practice to process the sequence in chunks; in other words, we read the first portion (i.e., a chunk) of the string, compute an intermediate internal state of the hash function, and continue from there without keeping the whole string in memory.

Consider a hash function H offering the *pause&resume* capability, and fix a constant block-size B: during the computation of $H(s)$, keep track of the internal state of the function (in a suitable *internal-states table*) at $H(s[\,..\,B])$, $H(s[\,..\,2B])$, $H(s[\,..\,3B])$, ..., and so on. The signature of any prefix $s[\,..\,k]$ can then be computed by loading the nearest internal state, $H(s[\,..\,\lfloor k/B \rfloor])$, and resuming the hash function from there, reading at most $B-1$ additional characters of s.

With this observation in mind, we can try out some of the fastest general-purpose hash functions available from off-the-shelf libraries to trade off a (maybe only theoretically) slower construction time for practically faster search queries.

6 Constant Time Substring Hashing

If we want to obtain linear construction time, we need to be able to compute the hash of *any substring* of a string s in constant time. For this purpose, we can use hash functions based on Cyclic Redundancy Checks (CRCs). CRCs are a widespread family of hash functions based on the division in the ring of polynomials with coefficients in the finite field $\mathbf{F}_2$, the field of integers modulo 2. In this field, addition and subtraction correspond to bitwise-xor and multiplication is a "carry-less product", which in binary arithmetic can be thought of as a repeated shift&xor operation similar to long multiplication. Hence, CRCs can be implemented using either hardware facilities (e.g., the PCLMULQDQ assembly instruction, i.e., the carry-less product of two 64-bit values) or with bitwise operations.

In the computation of CRCs, the whole string is interpreted as a long binary number whose digits become the coefficients of a polynomial; the signature is therefore the remainder of the polynomial modulo a fixed irreducible polynomial, the *generator polynomial*. More precisely, CRC hash functions are defined as:

$$H(s) = s(x)x^k \bmod g(x) \;\Leftrightarrow\; s(x)x^k = q(x)g(x) + H(s)$$

where $s(x)$ is the binary representation of the string s thought of as a polynomial in the variable x, $q(x)$ is the quotient polynomial, and $g(x)$ is the generator polynomial. The degree k of the generator polynomial $g(x)$ determines the size of the signature.

As part of the SSE4.2 extensions, modern CPUs provide full hardware support (i.e., using a single assembly instruction) for computing the 32-bit CRC used in the standard iSCSI; this CRC adopts a generator known as the Castagnoli polynomial [7].

CRCs are linear functions, and their internal state corresponds to the signature of the string up to that position. Given two strings a and b, we show how to compute efficiently $H(ab)$ starting from $H(a)$ and $H(b)$. In particular, we discuss how to append (internally) and prepend zero-bit strings to a signature; then, we can leverage the linearity properties to compose and decompose signatures ($\oplus$ denotes bitwise-xor):

$$H(ab) = H\big(a0^{|b|} \oplus 0^{|a|}b\big) = H\big(a0^{|b|}\big) \oplus H\big(0^{|a|}b\big).$$

We start by noting that $H\big(0^\ell s\big) = H(s)$ for any string s and for any integer ℓ: adding high-degree zero-coefficient terms to a polynomial does not affect its value. Appending zeroes, instead, consists in a polynomial multiplication between $s(x)$ (i.e., the polynomial of s) and x^ℓ. To compute $H\big(s0^\ell\big)$ we use the fact that internal states are signatures of prefixes. Instead of employing the shift&xor method, this operation is typically performed employing a lookup table for the linear transformation of appending power-of-two many zeroes to an internal state: by querying the table for every 1-bit in the binary representation of ℓ we can simulate encountering ℓ consecutive zeroes after reading the string.

Constant-time substring hashing is possible after precomputing the internal-state table as we discussed in Section 5, and a similar table for strings of the form 10^ℓ. Given two integers i and j such that $i < j$, the internal-state table of the string s enables us to compute $H(s[\,..\,i])$ and $H(s[\,..\,j])$ in constant time; then, we can apply the linear transformation that appends $j - i$ zero-symbols to $H(s[\,..\,i])$ and use linearity to calculate $H(s[i + 1\,..\,j])$ in constant time. However, we found that in practice the second table can be avoided since the cost of computing its values on-the-fly is logarithmic in ℓ.

7 Collision Detection during Fat Binary Search

As we already mentioned, we query the z-map with hashes of prefixes. It is thus possible that a prefix has the same hash value as a handle, even if it is a different string. If this happens, the exit node we obtain as a result of the fat binary search will be incorrect. To detect this behavior, we need to check after the search whether the node we found is *really* the exit node.

The standard *memcmp approach* consists of performing a full string comparison between the extent of the exit node and the pattern. This check cannot fail to detect an error.

An alternative, the *signature approach*, works as follows: instead of comparing strings character by character, in the case of CRC hashing we use constant time substring hashing to compare the pattern with the original text. The rationale is that since we can hash the original text and the pattern in constant time this check, albeit not exact, might be faster than the memcmp approach, and might provide an interesting tradeoff.

8 Experimental Evaluation

We ran our benchmarks on a 12th Gen Intel® Core™ i7-12700KF (Alder Lake) workstation, with 64 GiB of DDR4 RAM, GNU/Linux 6.1.14, and the GNU C Compiler (GCC) 13.0.1. We compiled the external dependencies (most notably, the hash function implementations) with optimizations enabled; note, in particular, that our architecture supports the BMI and the AVX2 instruction set. We tested two distinct CRC implementations using two different 32 bit generator polynomials. In particular, we used the software-based CRC implementation present in Zlib [12] and the hardware-based implementation in Facebook's Folly[1]. The hardware-based implementation uses the CRC and the PCLMULQDQ assembly instructions, with the optimizations discussed in [15].

The tests were performed on datasets from the standard Pizza & Chilli Corpus [9], which includes various types of text that have been selected as a representative sample of different applications of indexed text search. To account for the edge cases, we also included artificially generated texts made of random strings varying in alphabet size, and the highly repetitive Fibonacci words [4]. Each character in the randomly generated texts is extracted with equal probability from a set of 4 (rand-4) and 62 (rand-62) ASCII symbols. These random texts are used to reproduce some of the properties in the dna and in the english dataset in Pizza & Chilli. We report the data of the most important findings and present some more general considerations based on the insights given by the full set of experiments.

[1] https://github.com/facebook/folly/

8.1 Variants

We consider a number of implementations described in Table 1. Our baseline is enhanced, our implementation of the enhanced suffix array. The other variants are combinations of hash functions, either with memcmp or signature string comparisons.

Table 1 Naming conventions explained. B is the block size, a parameter that determines the size of a chunk in the internal-state table. S is the size (in bits) of the internal state of the hash function. For XXH3 we used the functions exposed by the library to save and load the internal state, which is suboptimal for our use case.

B	S	variant	meaning
—	—	enhanced	Abouelhoda *et al.*'s enhanced suffix array[1]
480 B	192	memcmp-wyhash	zuffix array with string comparison and wyhash
64 kB	4608	memcmp-xxh3	zuffix array with string comparison and XXH3
512 B	32	memcmp-folly	zuffix array with string comparison and hardware CRC
1 kB	32	memcmp-zlib	zuffix array with string comparison and software CRC
512 B	32	signature-folly	zuffix array with signature comparison and hardware CRC
1 kB	32	signature-zlib	zuffix array with signature comparison and software CRC

8.2 Statistics

In Table 2, we collected a series of statistics for each text. These statistics are meant to integrate those previously collected by the authors of the Pizza&Chili Corpus. Our additional data focus on suffix-tree structural properties and hash-function qualities: as we will see, our additional data make it possible to give a detailed explanation of the results of our experiments.

8.3 Suffix tree

By reporting the depth of the suffix tree, counting the number of internal nodes, and measuring the average length of name, handle, and extent of the internal nodes we aim to identify some of the variables needed to take advantage of the zuffification. In particular, we remark that the z-map speeds up the top-down navigation of suffix trees. The enhanced suffix array implements the suffix tree navigation in such a way that it induces a first-child next-sibling top-down tree traversal. Therefore, the enhanced suffix array can efficiently walk from the root to the exit when the alphabet is small, but we expect it to be slower on large alphabet texts. The running time of the fat binary search, instead, is not as directly dependent on the size of the alphabet.

8.4 Hash functions

We implemented the z-map using an open-addressing hash table with linear probing. For each hash function, we display the ratio number of distinct handles with equal signatures (*store collisions*). Note that with signatures of 64 bits or more we find no collisions. Moreover, collision problems can be partially mitigated with a few inexpensive runtime checks, preceding the techniques discussed in section 7. In a single search query, the z-map is expected to find nodes that lay on the path from the root to the exit node, whose handle length is the 2-fattest number in its skip interval. With the implicit representation of the tree provided

by the enhanced suffix array, ancestor queries are trivially answered in constant time by comparing lcp intervals [1]. This consideration will intuitively lead us to the conclusion that collisions will not play the biggest role in the choice of hash functions, an intuition that will be confirmed by the experimental results.

8.5 Space

The number of internal nodes also counts how many handles are stored in the z-map and it can be used to estimate the space consumed by the data structure. Consider a standard (non-compressed) implementation of the enhanced suffix array. Given a text of length n, let $w \geq \lceil \lg n \rceil$ be the size in bits of an unsigned integer large enough to hold indices of the text, and let B be the block size of a hash function with an internal state of size S. We denote with slack$(-)$ the function that determines the size of the hash table given the number of internal nodes; in our implementation, tables are sized as powers of two, and they are never filled more than 2/3. The space used to implement a zuffix array is thus given by the sum of the space occupied by the enhanced suffix array (suffix array, lcp array, additional array),

$$\begin{aligned} \text{space(sa)} &= nw \\ \text{space(lcp)} &= (n+2)w \\ \text{space(add)} &= nw, \end{aligned}$$

by the space occupied by the z-map (hash and node),

$$\text{space}(Z) = (2w + (c+\varepsilon)\log n) \cdot \text{slack}(\#\text{ internal nodes})$$

and by the space occupied by the internal-state table of the hash function in the signature case

$$\text{space}(H) = nS/B.$$

The block size and the state size used in our implementation are reported in Table 1 and, in our benchmarks, we set w to 64. While performing string matching, the zuffix array consumes additional mS/B space for the internal-state table of the pattern. We tuned the block size of the hash function for speed and we obtained internal-state tables up to 32 times smaller than the strings.

8.6 Performance

A well-known assumption in stringology is that the pattern is orders of magnitude shorter than the text. From a theoretical standpoint, this folklore assumption can be justified by analyzing the suffix tree of the text, as we did in Table 2. In a suffix tree, the string-matching problem is, fundamentally, an exit node search followed by a string comparison. The names of the leaves provide a good description of the longest "meaningful" patterns we may search for: all the strings that are longer than the maximum name have zero or only one occurrence in the text. If two strings are longer than the maximum name in the suffix tree and if one of them is the prefix of the other, then the running time of the string-matching algorithm differs by only a (longer) string comparison. By considering only the patterns whose length is smaller than the maximum name length, we aim to focus only on the most significant portion of the performance plot.

The texts from the Pizza&Chilli Corpus come from real sources, and they are meant to be representative of real-life applications of text indexing. The two random texts and the

Table 2 Statistics for each text dataset.

	dna	proteins	dblp-xml	sources	english	rand-4	rand-62	fibonacci
σ	4	27	97	230	239	4	62	2
size	200 MiB	200 MiB	200 MiB	200 MiB	200 MiB	200 MiB	200 MiB	8.80 MiB
suffix tree depth	165	391	123	3237	122	19	8	48
# internal nodes	142 M	268 M	100 M	131 M	129 M	130 M	35 M	9 M
avg$\{\|n_\alpha\|\}$	18	252	37	149	124	13	5	1603384
avg$\{\|h_\alpha\|\}$	64	380	40	452	10732	13	5	2017069
avg$\{\|e_\alpha\|\}$	81	446	43	566	15171	13	5	2435423
construction time	12.10 s	17.34 s	9.97 s	10.85 s	11.72 s	12.15 s	12.30 s	280.94 ms
suffix tree space	4.7 GiB	4.7 GiB	4.7 GiB	4.7 GiB	4.7 GiB	4.7 GiB	4.7 GiB	211.20 MiB
wyhash (64 bit pause&resume)								
zuffification	4.73 s	6.60 s	3.80 s	5.76 s	47.6 s	4.23 s	1.71 s	8 m 24 s
z-map space	6 GiB	6 GiB	6 GiB	6 GiB	6 GiB	6 GiB	1.5 GiB	384 MiB
store collisions	0	0	0	0	0	0	0	0
xxh3 (128 bit pause&resume)								
zuffification	5.02 s	6.71 s	4.39 s	5.64 s	34.8 s	4.24 s	1.60 s	5 m 22 s
z-map space	8 GiB	8 GiB	8 GiB	8 GiB	8 GiB	8 GiB	2 GiB	512 MiB
store collisions	0	0	0	0	0	0	0	0
folly (32 bit CRC hardware)								
zuffification	14.2 s	12.2 s	10.1 s	11.2 s	12.7 s	13.3 s	4.80 s	526 ms
z-map space	5 GiB	5 GiB	5 GiB	5 GiB	5 GiB	5 GiB	1.25 GiB	320 MiB
store collisions	1.62%	0.65%	1.16%	1.52%	1.50%	1.52%	0.22%	0.22%
zlib (32 bit CRC software)								
zuffification	62.9 s	55.0 s	43.5 s	53.0 s	56.2 s	58.4 s	16.5 s	4.58 s
z-map space	5 GiB	5 GiB	5 GiB	5 GiB	5 GiB	5 GiB	1.25 GiB	320 MiB
store collisions	1.71%	0.65%	1.16%	1.52%	1.50%	1.74%	0.22%	0.11%

Fibonacci words dataset are artificial and they are best used as a means of comparison. Each text is trimmed to 200 MiB, so the results are normalized in size; the only exception is the text for the Fibonacci words, whose length is F_{35}. The symbols are always considered to be byte-based, so in the Pizza&Chilli Corpus we do not split English words into terms and the gene DNA nucleotides (or any unknown choice among them) are represented with their standard ASCII characters.

8.7 Construction

Albeit the general-purpose hash functions with the *pause&resume* capabilities discussed in Section 5 induce a suboptimal construction time, in practice the best functions available from off-the-shelf libraries are faster than CRCs, while also offering a wider output, leading to fewer collisions. In Table 2 we report the running time of zuffification. This result can be compared with the construction time of the enhanced suffix array, also reported in the table. For the construction of the suffix array we are using Ilya Grebnov's `libsais` [13] compiled with OpenMP support. To the best of our knowledge, this is the fastest suffix array construction algorithm currently available. Note that our current implementation of zuffification is not multi-threaded, so the work done to construct the z-map is much smaller than the work required to build the enhanced suffix array. The size of the z-map is also reported in the table: it is comparable to the size of the enhanced suffix array for 32-bit CRCs, it becomes slightly larger in the case of 64-bit hashes, and it is almost twice as big for 128-bit hashes. The only exception is rand-64 because the number of internal nodes in that case is very small.

Fibonacci words contain, asymptotically, the maximum number of repetitions that may be contained in any string. Fibonacci words are a worst-case example for the construction algorithm and in the fibonacci column of Table 2 we can see how the theoretical results are reflected in practice: zuffification under the general-purpose *pause&resume* hash functions can

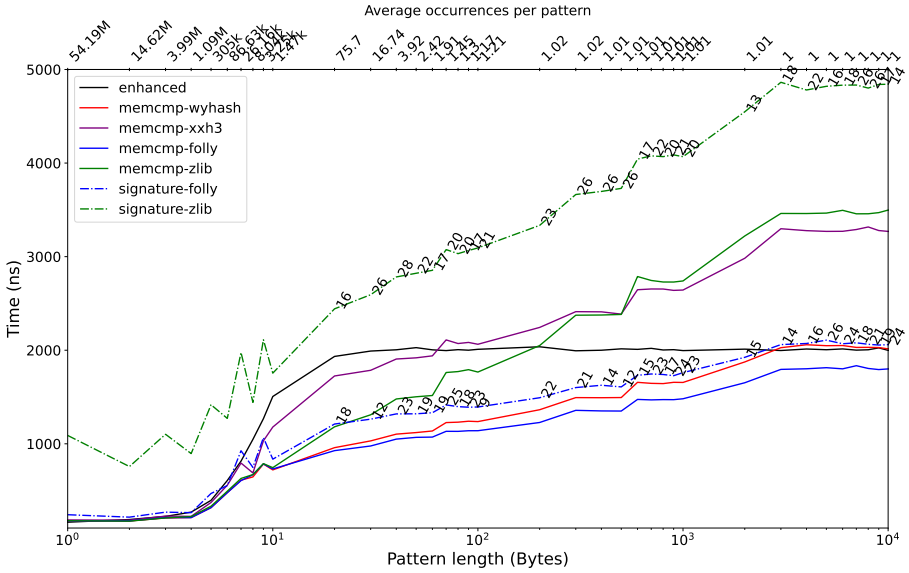
be several times slower than CRCs. Nevertheless, in all the other (more realistic) scenarios that we have studied, the general-purpose hash functions are competitive and sometimes outperform the theoretically faster hash functions offering constant-time substring hashing. This result is mostly a consequence of the statistical properties of the text and it is not just a matter of its size. The fibonacci corpus maximizes the length of the names; thus, zuffification can be significantly slower than in other corpora even if its text is much smaller.

Table 3 Raw performance (in nanoseconds) of string search by pattern size.

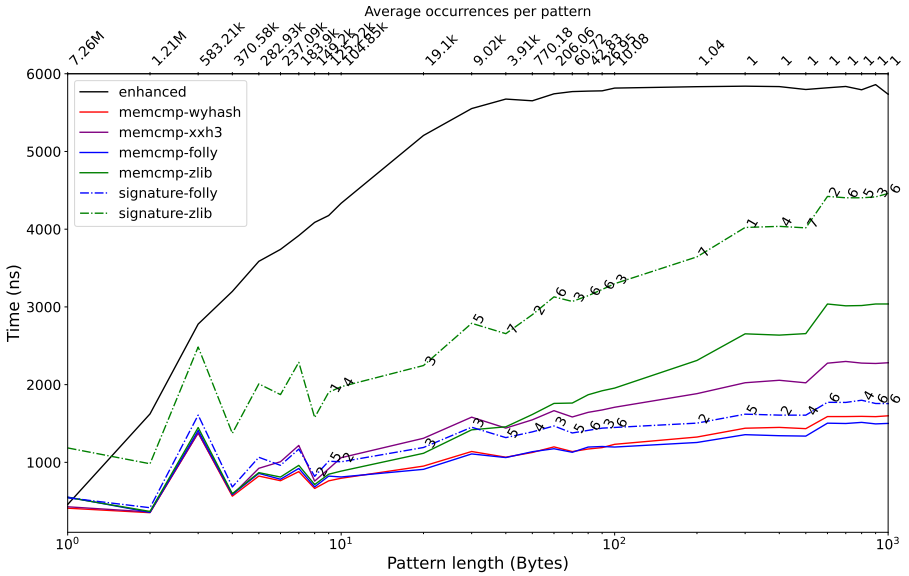
	size	enhanced	mem-wyhash	mem-xxh3	mem-folly	mem-zlib	sgn-folly	sgn-zlib
dna	10^1	1505	**720**	1178	731	745	835	1755
	10^2	2009	1237	2063	**1139**	1766	1393	3092
	10^3	1996	1655	2643	**1481**	2740	1760	4068
	10^4	1998	2014	3269	**1800**	3497	2059	4842
proteins	10^1	3153	**814**	1365	848	973	1105	2190
	10^2	3267	1125	1710	**1116**	1637	1350	2895
	10^3	3326	1587	2345	**1487**	2772	1758	4075
	10^4	3323	2495	3872	**2180**	4851	2447	6175
dblp	10^1	4333	**795**	1059	811	887	1007	1973
	10^2	5815	1232	1708	**1196**	1955	1448	3298
	10^3	5736	1598	2281	**1500**	3037	1755	4459
sources	10^1	6810	**850**	1255	891	994	1103	2131
	10^2	7300	**1220**	1760	1229	1884	1484	3108
	10^3	7379	1695	2382	**1605**	2959	1858	4288
	10^4	7352	2612	3915	**2337**	5068	2588	6389
english	10^1	6137	**799**	1236	859	931	1029	2100
	10^2	6365	1203	1888	**1166**	1760	1445	3087
	10^3	6424	1625	2437	**1535**	2819	1825	4186
	10^4	6458	2428	3735	**2219**	4843	2477	6131
rnd-4	10^1	1459	**716**	1174	721	737	835	1748
	10^2	1732	931	1709	**856**	1178	1210	2491
rnd-62	10^1	5033	742	1228	**723**	809	916	1857
	10^2	5028	748	1205	**717**	821	918	1835
fibo	10^1	**160**	163	186	161	174	236	1301
	10^2	212	217	273	**207**	508	285	1935
	10^3	**325**	536	712	570	1575	815	3659
	10^4	**638**	1060	1473	1083	3110	1213	4844

8.8 Search queries

In Table 3 we report the query performance of all datasets for medium-length patterns, whereas in Figures 2–4 we show the performance of each variant under four significantly different scenarios and a wide range of lengths. Each plot reports the running time of search queries. Expectedly, the running time increases with the length of the pattern. For each pattern size, we generated 10 000 patterns extracted from random positions of the text and we measured the running time of each variant using standard benchmarking techniques. The running time reported in the plots is meant to be representative of that of a single search query; in particular, we avoided the potential caching speedups that may occur when several operations of the same kind happen in near succession. We remark that zuffix arrays are expected to access the underlying suffix tree only once, and thus the underlying tree could even be stored in external memory, making the space occupation of the whole structure a less daunting obstacle than what might appear from the raw numbers in Table 2.

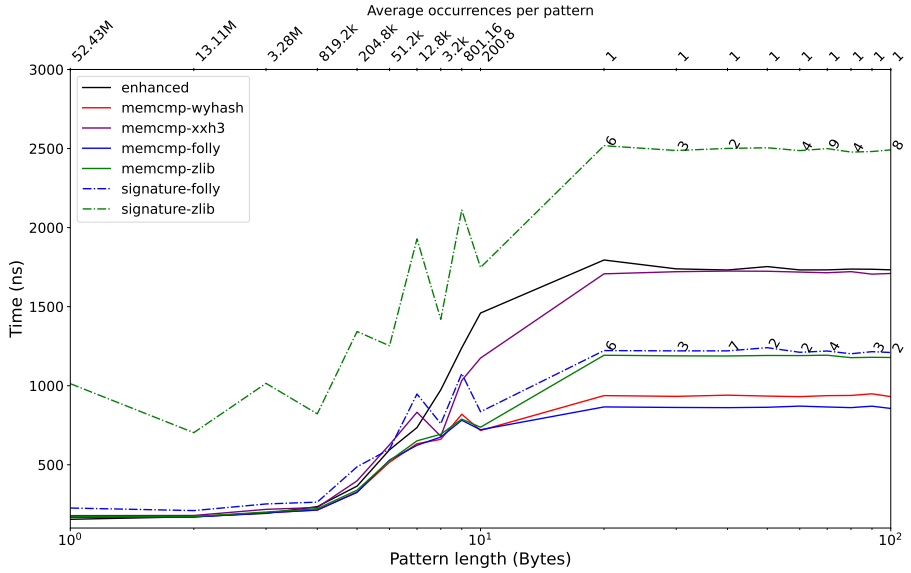


(a) dna: highly-repetitive small alphabet.

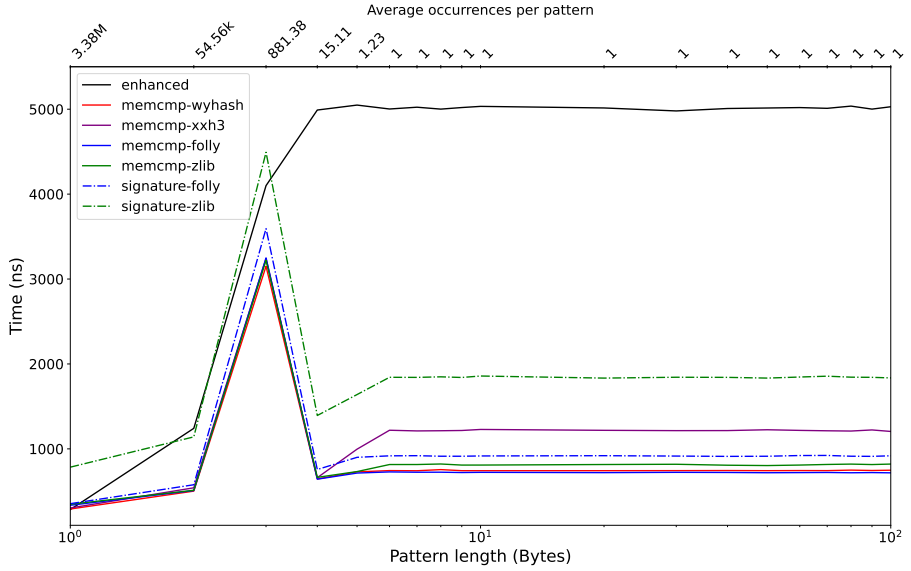


(b) dblp-xml: highly-repetitive large alphabet.

Figure 2 Raw performance of string search by pattern size.

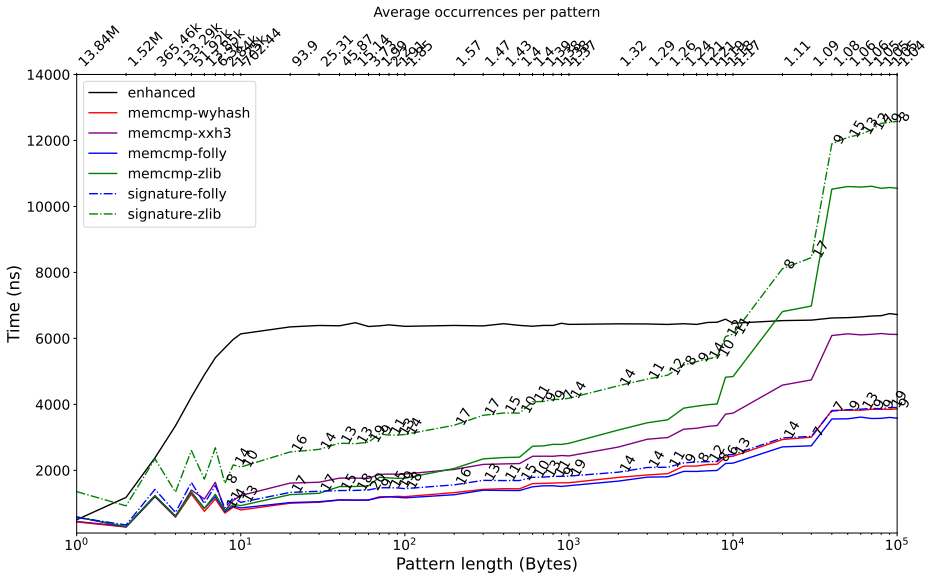


(a) rand-4: barely repetitive small alphabet.

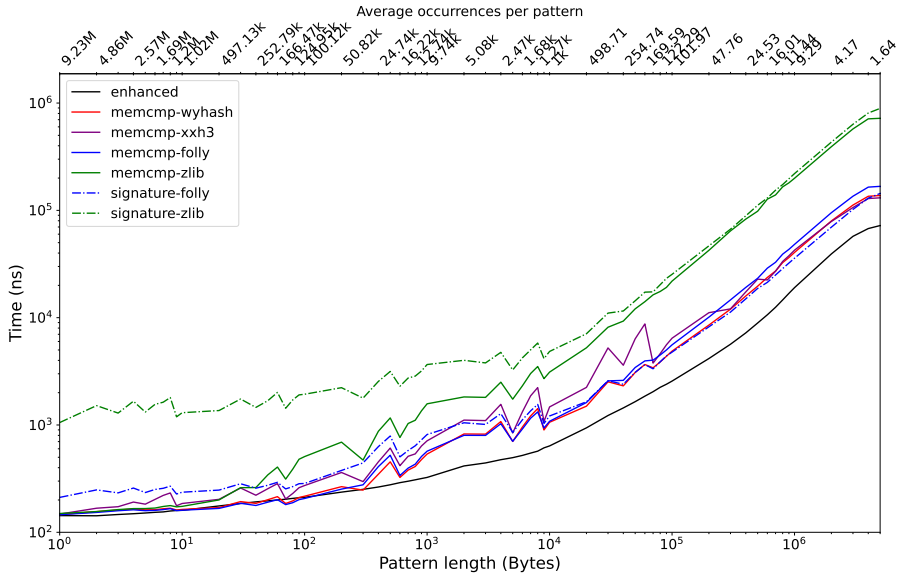


(b) rand-62: barely repetitive large alphabet.

Figure 3 Raw performance of string search by pattern size.



(a) english: natural language.



(b) fibo: Fibonacci words.

Figure 4 Raw performance of string search by pattern size.

The signature variants may return false positives: if there are any for some pattern, we report their number on the plot (the time required to check for false positives is included in our benchmarks). Counting the false positives in signature variants is also indicative of the performance of their memcmp counterpart: when the z-map fails to find the exit node, the string search algorithm (implicitly) falls back to a regular (non z-fast, but starting from the root) search in the underlying suffix tree. Even in more extensive experiments, the average numbers of false positives in the CRC variants never exceed the 0.3% of the queries, suggesting that this worse-case search happens quite rarely, as we can also evince from the plots, and that 32-bits hash functions might be sufficient in most applications.

We search the same patterns for each variant; parallel to the length of the patterns, we report the average number of occurrences of each pattern as a different characterization of complexity: the running time increases as the size of the pattern grows, but when the average number of occurrences approaches one the hardness of the search problem remains the same, independently of the pattern length, and the only additional cost is the final check. Thus, after this point we do not perform tests with longer patterns: this strategy is evident in the random datasets, in which short patterns are sufficient to identify completely a single result.

The running time of our baseline (the enhanced suffix array) is mainly dependent on the alphabet size: the larger the alphabet, the longer it takes to walk down the (implicit, first-child next-sibling) suffix-tree representation. The performances of the enhanced suffix array in rand-62 and in english are similar, because both texts can be thought of as being mostly made of Latin uppercase and lowercase letters, and few other symbols – despite the alphabet in the english corpus is much larger (see Table 2), most of the other symbols rarely occur.

As we previously noted, while all the zuffix-array variants have an underlying enhanced suffix array, the size of the alphabet has little to no influence on their performance, as after the fat binary search we access a single node of the suffix tree. Therefore, zuffification can be used to improve the performance of enhanced suffix arrays on large alphabets.

The performance of the zuffix arrays is mostly dependent on the average length of the handle of the exit node. We can see indeed that all the zuffix-array variants have a similar running time across all datasets, except in the case of Fibonacci words, where the small alphabet and the enormous average handle length makes fat binary search less effective.

We can observe that zuffix arrays, at least in their fastest variants (hardware CRC and wyhash), are always faster than their non-accelerated variant (again, except in the case of Fibonacci words). Despite the enhanced suffix array being a very performant data structure, especially in small alphabet texts, zuffix arrays provide impressive speedups. In a small alphabet text like dna, the zuffix array is up to twice as fast as the enhanced suffix array. In a larger text, like dblp-xml and english, the zuffix array is up to 7 times faster than its baseline. Even when the patterns have only one occurrence in the text, the zuffix array is 1.5 to 3.5 times faster than the enhanced suffix array.

There is of course a price to pay: the zuffix array is comparable in size to the underlying enhanced suffix array in the CRC version, so it doubles the memory requirements. Nonetheless, for large alphabets we obtain a speedup of almost one order of magnitude, which we believe is a very interesting tradeoff.

9 Lessons Learned

- *The z-map can make searches several times faster for medium-length patterns.* As one can see from Table 3 and Figure 2–4, searches on such patterns are up to 7 times faster than on an enhanced suffix array. As the pattern length grows, however, the cost of memory access becomes dominant, so in the case of highly repetitive text with a small alphabet and long patterns the z-map theoretical improvement becomes unnoticeable.
- *Hardware-supported CRCs are ideal hash functions.* Software-based CRCs are too slow (indeed, by far the worst-performing hashing strategy). Ideally, a 64-bit wide hardware-supported CRC would induce fewer collisions, but even in the presence of collisions (which sometimes requires a full comparison of the pattern) hardware-supported CRCs are still the fastest option.
- *Standard hash functions are sufficient for practical purposes.* While CRCs are ideal in general, unless we consider pathological counterexamples, such as our Fibonacci dataset, using a standard, highly optimized hash function yields good results and sometimes provides a shorter construction time than CRCs. Construction time is theoretically quadratic, but on realistic datasets node names (and thus handles) are quite short, and modern hash functions are very fast.
- *The signature approach is not useful.* For the same reason (short names) the signature approach is (maybe surprisingly) slower than the memcmp approach. The problem here is that when performing the comparison, one has to hash the entire pattern, and since precomputed hash states are generated lazily, almost all the work is done at the time of the comparison, so in reality the check is closer to use time that is linear in the pattern length, rather than constant; at that point, memcmp is faster.
- *Size depends on structure.* The size of the z-map depends heavily on the structure of the text, in particular for large alphabets: while the number of internal nodes of a binary trie is equal to the number of leaves minus one, in the case of large alphabets it depends strictly on the structure of the text. The z-map is smaller in the case of few internal nodes with many children, which is exactly the case in which the enhanced suffix array is slower.

10 Conclusions

We have presented a detailed and carefully engineered implementation of zuffix arrays based on enhanced suffix arrays. We have shown that the z-map can improve by several times the performance of an enhanced suffix array on patterns of medium length, reaching a sevenfold speed increase in the case of large alphabets. While the space occupied by the z-map is comparable to that of the enhanced suffix array, we believe that if space is available zuffix arrays provide an interesting tradeoff between time and space, providing the practically fastest pattern lookup time.

As we remarked in the introduction, the z-map should be thought of as a speed-up gadget that can be superposed to different trie-based data structures: in the future, we plan to experiment with its application to further underlying data structures, provided that they make it possible to navigate the suffix tree.

Another interesting line of research is that of reaching different points of the space-time tradeoff by pruning the longer handles of the z-map. Indeed, the z-map works as long as it is built on a prefix of the suffix tree, but in that case, completing the search might require more than one access to the suffix tree, as it is in the case of this paper. Depending on the underlying structure, which could be, for example, compressed but very slow, choosing different prefixes would provide different tradeoffs between space and time.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

2 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 785–794, New York, 2009. ACM Press.

3 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In Edgar Chávez and Stefano Lonardi, editors, *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2010.

4 Jean Berstel. Fibonacci words—A survey. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 13–27. Springer–Verlag, 1986.

5 Paolo Boldi and Sebastiano Vigna. Kings, name days, lazy servants and magic. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

6 Manuel Cáceres and Gonzalo Navarro. Faster repetition-aware compressed suffix trees based on block trees. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 434–451. Springer, 2019. doi:10.1007/978-3-030-32686-9_31.

7 Guy Castagnoli, Stefan Brauer, and Martin Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Trans. Commun.*, 41(6):883–892, 1993. doi:10.1109/26.231911.

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

9 Paolo Ferragina and Gonzalo Navarro. The pizza & chili corpus, 2007. URL: http://pizzachili.dcc.uchile.cl/texts.html.

10 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009. doi:10.1016/J.TCS.2009.09.012.

11 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, January 2012.

12 Jean-Loup Gailly and Mark Adler. Zlib compression library. Technical report, Apollo - University of Cambridge Repository, 2004. URL: http://www.dspace.cam.ac.uk/handle/1810/3486.

13 Ilya Grebnov. libsais. https://github.com/IlyaGrebnov/libsais, 2021.

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

15 Shay Gueron and Michael E. Kounavis. Efficient implementation of the Galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010. doi:10.1016/J.IPL.2010.04.011.

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

17 Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

18 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

19 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

20 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

21 Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In Edgar Chávez and Stefano Lonardi, editors, *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 322–333. Springer, 2010. doi:10.1007/978-3-642-16321-0_34.

22 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

23 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

24 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.