Kings, Name Days, Lazy Servants and Magic

Paolo Boldi
Università degli Studi di Milano, Italy
paolo.boldi@unimi.it
https://orcid.org/0000-0002-8297-6255

Sebastiano Vigna
Università degli Studi di Milano, Italy
sebastiano.vigna@unimi.it
https://orcid.org/0000-0002-3257-651X

Abstract

Once upon a time, a king had a very, very long list of names of his subjects. The king was also a bit obsessed with name days: every day he would ask his servants to look the list for all persons having their name day. Reading every day the whole list was taking an enormous amount of time to the king's servants. One day, the chancellor had a magnificent idea: he wrote a book with instructions. The number of pages in the book was equal to the number of names, but following the instructions one could find all people having their name day by looking at only a few pages – in fact, as many pages as the length of the name – and just glimpsing at the list. Everybody was happy, but in time the king's servants got lazy: when the name was very long they would find excuses to avoid looking at so many pages, and some name days were skipped. Desperate, the king made a call through its reign, and a fat sorceress answered. There was a way to look at much, much fewer pages using an additional magic book. But sometimes, very rarely, it would not work (magic does not always work). The king accepted the offer, and name days parties restarted. Only, once every a few thousand years, the magic book fails, and the assistants have to go by the chancellor book. So the parties start a bit later. But they start anyway.

2012 ACM Subject Classification Theory of computation → Sorting and searching

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

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.10

1 Introduction

From what we can ascertain reading the enthusiastic reports of the contemporary historians, the chancellor probably stumbled into an early version of *suffix arrays* [11]. His book might have contained a table of the initial points in the list for every name, sorted lexicographically by suffix. Indeed, the suffix array of a string s over an ordered alphabet Σ of σ elements, in modern terms, is simply the array of the starting points of the string $s\$$ (where $\$$ is a character larger than any character in Σ) sorted lexicographically by the corresponding suffix. Suffix arrays are an extremely effective way of looking for all 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. In modern days, a large body of research has gone into building and representing suffix arrays efficiently (e.g., in compressed form) [6, 4, 13, 10, 7].

In fact, at the price of an additional (and very compressible) array of integers a suffix array can represent implicitly the suffix tree associated with the string s [1]. While asymptotically the two approaches give the same bounds, we know that managing a billion nodes or three arrays of a billion integers is an entirely different business, especially if you are a poor servant that lives on bread and water.

So, why were the servants still unhappy? Well, since the pages to look at were as many as the letter in the names, with long names they had to jump through several pages of the book and of the list quite at random. The book was heavy, and the list too. Looking at consecutive pages was easy, jumping around much less.

Here is when the fat sorceress came in: she said that with an additional book and additional instructions, looking at a number of pages equal to the *logarithm* of the length of the name would have been sufficient to recover the same information, together with a few scans on the list. The only problem is that magic is tricky and once in a while (although very rarely) it would not work: if that happens, though, sevants would realize that something went wrong, and they could still go the old way.

Through a thorough research we have been able to reconstruct the spell. What the fat sorceress probably discovered is a way to apply *fat binary search*, the main ingredient of *z-fast tries* [2], to a suffix array.[1] To use fat binary search, one stores some additional linear-space information, the *z-map*. Then, given a pattern p of length m, one first preprocesses p in time $O(1 + (m \log \sigma)/w)$, and then performs $O(\log m)$ search steps, each accessing a constant amount of information. Then follows a verification phase, which accesses 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. Thus, in modern-day parlance, fat binary search 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.

2 Notation and Tools

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, $x[|x| \,..\,] = \varepsilon$.

We analyze our algorithms on a unit-cost word RAM with word size w in the cache-oblivious model [5]. 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 an 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 is $O(1 + N/B)$.

We measure space in words. Thus, we will say that a suffix array takes linear space, even if it needs $O(n \log n)$ bits. A more detailed analysis can be performed when specific instances of the various support structure have been instantiated (e.g., a compressed vs. non-compressed lcp array).

[1] We should mention here that trying such a spell was suggested to the sorceress by the great sorcerer of sorcerers, Ricardo Baeza-Yates.

3 Static Z-Fast Tries on Arbitrary Languages

Fat binary search appeared for the first time in the context of *probabilistic static z-fast tries* [2], which were originally introduced for prefix-free binary languages. Their main purpose was to assign buckets to a larger set of strings. The basic idea is that of enriching a standard compacted trie with a kind of *acceleration map*, the *z-map*, which makes it possible navigate the trie quickly (i.e., using a number of accesses to the map logarithmic in the length of the search string).

Subsequently, fat binary search was applied to *dynamic z-fast tries* [3], again based on prefix-free binary strings. Dynamic z-fast tries exist in two versions: an *exact* version and a *signature-based* version. In the first case, we store a z-map from strings to nodes, in the second case a z-map from string *signatures* to nodes, and we have to handle collisions and false positives.

In this section we start introducing exact *static* z-fast tries on prefix-free languages L on a general alphabet Σ, and show how they can be used to solve the following problem: is p the prefix of some element of L? We describe the algorithms in the exact setting, adding assertions that may fail in the signature-based setting.

3.1 Compacted tries

A *compacted trie* [9] 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* $e_\alpha \in \Sigma^*$ 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[2] $T(L)$ 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 trie with the nomenclature just introduced (and some more that will be introduced in the following).

3.2 Exit nodes

Given a compacted trie over Σ, and given a string p, we let $\operatorname{exit}(p)$ be the exit node of p, that is, the only node α such that $n_\alpha \preceq p$ and for every other node α' if $n_{\alpha'} \preceq p$ then $n'_\alpha \preceq n_\alpha$. In other words, it is the node whose name is the longest possible prefix of p. (See Figure 1 for an example of an exit node). Moreover, we call *parex node of* p the *parent* of the exit node of p, or a special symbol $\perp$ if the exit node of p is the root (note that the parex of p is the node with the longest extent that is a *proper* prefix of p).

It is worth stating the following property of exit nodes:

[2] In [2], we studied compacted tries only for the binary case (i.e., $\Sigma = 2$).

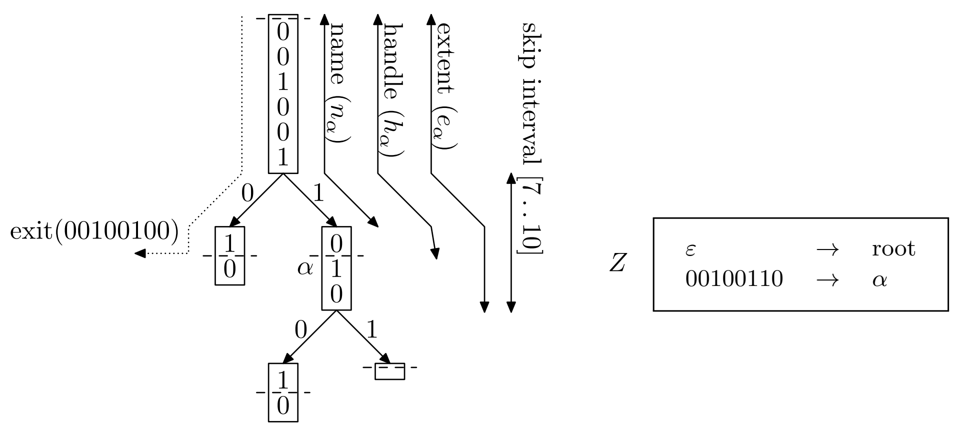

Figure 1 (above) The compacted trie $T(L)$ (with the corresponding nomenclature) and its z-map. Here $L = \{001001010, 0010011010010, 00100110101\}$.

▶ **Lemma 1.** *For a given string p, $\operatorname{exit}(p)$ is the only node γ such that $n_\gamma \preceq p$ and one of the following mutually exclusive properties holds:*

- *$p \preceq e_\gamma$;*
- *$e_\gamma \prec p$ and γ does not have a $p[|e_\gamma|]$ child;*
- *p and e_γ are $\preceq$-incomparable.*

Proof. The only remaining case is that $e_\gamma \preceq p$ with γ having a $p[|e_\gamma|]$-child: but in this case, that child would have a name that is still a prefix of p, longer than n_γ. ◀

Another easy consequence of the definition is the following:

▶ **Proposition 2.** *Let $L \subseteq \Sigma^*$ and consider the trie $T(L)$. A string $p \in \Sigma^*$ is a prefix of some element of L if and only if $p \preceq e_{\operatorname{exit}(p)}$. Moreover, if the latter happens then the set*

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

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

In the example of Figure 1, $p = 00100100$ is not the prefix of any of the strings in the language. The name of its exit node $\operatorname{exit}(p)$ (the leftmost child of the root in the figure) is 0010010, which is in fact a prefix of p. Yet the extent of $\operatorname{exit}(p)$ is 001001010 of which p is not a prefix. If we had taken $p' = 00100110$ we would have exited at the rightmost child of the root, whose extent 0010011010 has p' as prefix: in fact, p' is the prefix of two of the elements of L, corresponding precisely to the rightmost two leaves in the trie.

A final important remark is the following: if p is *not* a prefix of an element of L, according to Proposition 2, $p \not\preceq e_{\operatorname{exit}(p)}$. But more than this is true: $p \not\preceq e_\alpha$ for all nodes α (for otherwise, a fortiori, p would be a prefix of the extent of a leaf).

3.3 Static z-fast tries

Let us assume that we have built the trie $T(L)$ for a given language L of size n. Proposition 2 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 $\operatorname{exit}(p)$ and then to check whether p is a prefix of its extent or not. Moreover, the second part of the statement suggests which elements of L have p as prefix.

Algorithm 1 Querying the z-fast trie using fat binary search. Given a string p, it will return either $\operatorname{exit}(p)$ or $\operatorname{parex}(p)$.

```
Input: a string $p$ of length $m$
Output: either $\operatorname{parex}(p)$ or $\operatorname{exit}(p)$
$\ell, r \leftarrow 0, m$
while $\ell \leq r$ do
  $f \leftarrow$ the 2-fattest number in $[\ell \mathinner{..} r]$
  $\beta \leftarrow Z(p[0 \mathinner{..} f-1])$
  if $\beta \neq \perp$ then
    $\ell \leftarrow |e_\beta| + 1$
    $\gamma \leftarrow \beta$
  else
    $r \leftarrow f - 1$
  end if
end while
return $\gamma$
```

Locating $\operatorname{exit}(p)$ can be done trivially in $O(m\sigma)$ steps, going down in the trie starting from the root. In [2] we suggest an alternative, faster solution that needs an additional data structure, called *z-map*. We briefly recall the idea.

▶ **Definition 3** (2-fattest numbers and handles)**.** The *2-fattest number* of an interval $[a \mathinner{..} b]$ of non-negative integers is the unique integer in $[a \mathinner{..} b]$ that is divisible by the largest power of two, or equivalently, that has the largest number of trailing zeroes in its binary representation. The *handle* h_α of a node α of a trie is the prefix of e_α whose length is 2-fattest in $[|n_\alpha| \mathinner{..} |e_\alpha|]$ (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 ending just above the dotted lines you can see in each node.

▶ **Definition 4** (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.

In the example of Figure 1, there are only two internal nodes, so the map contains only two pairs. 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 the key is not in the dictionary.

The usefulness of the z-map is made evident by Algorithm 1, which takes as input a string $p \in \Sigma^*$ and outputs a node of the trie which is either $\operatorname{exit}(p)$ or $\operatorname{parex}(p)$: it is a general-alphabet version of the classic fat binary search [2], and in fact, it is exactly identical to the binary version, since the alphabet has no role in such searches. The proof from [2] goes along in the same way.

▶ **Theorem 5.** *Algorithm 1 is correct and its loop is executed at most* $\log m$ *times; in particular, the z-map is accessed at most* $\log m$ *times.*

Note that fat binary search does not use the trie structure: it only queries the z-map, each time using a prefix of p, and computes possibly the length of the extent of a node returned by the z-map.

If our purpose is determining whether p is the prefix of some element of L, we can start with Algorithm 1, but then we need two things: first we must understand whether the

Algorithm 2 Given a string p, it will return either $\mathrm{exit}(p)$ or $\perp$, depending on whether p is the prefix of a string in L or not.

```
Input: a string $p$
Output: if $p$ is the prefix of an element of $L$ then $\mathrm{exit}(p)$, otherwise $\perp$
$\gamma \leftarrow$Algorithm 1 on $p$
assert $n_\gamma \preceq p$
if $p \preceq e_\gamma$ then
  return $\gamma$ /* $\gamma$ is the exit node */
else
  if $p$ and $e_\gamma$ are $\preceq$-incomparable then
    return $\perp$ /* $\gamma$ is the exit node, but $p \not\preceq e_{\mathrm{exit}(p)}$ */
  else
    /* necessarily $e_\gamma \prec p$ */
    if $\gamma$ has a child labelled $p[|e_\gamma|]$ then
      let $\gamma'$ be the child /* $\gamma'$ is the exit node */
      if $p \preceq e_{\gamma'}$ then
        return $\gamma'$
      else
        assert $p$ and $e_{\gamma'}$ are $\preceq$- incomparable or $\gamma'$ does not have a $p[|e_{\gamma'}|]$-child
        return $\perp$
      end if
    else
      return $\perp$ /* $\gamma$ is the exit node, but $p \not\preceq e_{\mathrm{exit}(p)}$ */
    end if
  end if
end if
```

algorithm returned the exit node or its parent (and in the latter case we must go down to the actual exit node), and second we need to use Proposition 2 to determine if $p \preceq e_{\mathrm{exit}(p)}$ or not. Algorithm 2 does both things at the same time.

▶ **Theorem 6.** *Algorithm 2 is correct; moreover, if the length of the strings in L is bounded by $O(w/\log\sigma)$, the algorithm uses $O(\log m + \sigma)$ time and I/Os.*

Proof. First of all, note that since $n_\gamma \preceq p$ (as in the first **assert**), one can determine if $p \preceq e_\gamma$ just by comparing $p[|n_\gamma| \,..\, |e_\gamma|]$ and c_γ. A similar consideration is true (later on in the algorithm) for deciding if $p \preceq e_{\gamma'}$.

Let now γ be the output of Algorithm 1; γ is either the exit node or the parex: in either case $n_\gamma \preceq p$ (which justifies the first assertion). If $p \preceq e_\gamma$, necessarily γ is the exit node and moreover p is a prefix of some element of L by Proposition 2. If p and e_γ are incomparable, once more γ is the exit node (because none of the children of γ can have a name that is a prefix of p), but (again using Proposition 2) we must return $\perp$. The last case is that $e_\gamma \prec p$. If γ has no child with label $p\big[|e_\gamma|\big]$, for the same reasons as in the last case, γ is the exit node but we must return $\perp$. Otherwise, the child with label $p\big[|e_\gamma|\big]$, say γ', is the real exit node. We must continue to check if $p \preceq e_{\gamma'}$ or not, but we can limit the check to the part of the string p that was not checked so far. The last assertion states that γ' is indeed the exit node.

For the complexity statement, the call to Algorithm 1 requires time $O(\log m)$ and the same amount of I/Os to query $Z(-)$ (every access to $Z(-)$ is done in constant time because of the assumption about the length of the strings in L). The comparison between p and

e_γ (and, later, $e_{\gamma'}$) requires time $O(1 + (m \log \sigma)/w)$ and $O(1 + (m \log \sigma)/B)$ I/Os (as we observed above, in both cases we are comparing a substring of p with the compacted path of a node), but $m = O(w/\log \sigma)$ so these quantities are both $O(1)$. The only case in which Algorithm 1 needs to access the trie structure is when it needs to find the child of γ with label $p\big[|e_\gamma|\big]$, which can be done in time $O(\sigma)$. ◀

It is worth noting that Algorithm 2 performs at most one access to the trie structure in case it needs to enumerate the children of γ. Moreover, the comparisons between p and other strings requires at most σ scans of overall m characters.

We also remark that if the z-map is modified to include also the handles of the leaves, Algorithm 1 on a string in L never outputs the parex, which means that Algorithm 2 needs just $O(\log m)$ time and I/Os to solve a membership query.

3.4 Signature-based static z-fast tries

If the length of the strings in L is not bounded by $O(w/\log \sigma)$, the map $Z(-)$ described in the previous section 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*: 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 fat binary searches return the correct result only with high probability.

Note that by using hashes of size $(c + \varepsilon) \log n$, $c \geq 2$, 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 a collision when t elements are extracted from a set of u with replacement is well approximated by $1 - e^{-t^2/2u}$, which means that with the choice above the probability of having a hash collision between distinct handles is at most

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

Once 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 positives, which by the union bound happen 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 have to compute the hash of a potentially long prefix: to 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. We will see that even stronger properties will be needed in Section 5.

Finally, the signature-based version needs that besides c_α, also n_α can be accessed in constant time from α. This can be obtained in different ways, but usually the simplest one (and the one used by the dynamic z-fast trie) is to store in α a pointer to a suitable element of L. Another possibility is to store explicitly e_α.

Due to signature collisions and false positives, Algorithm 1 may output a node γ that is neither the exit node nor the parex. Let us see how this fact impacts on Algorithm 2; looking back at Lemma 1, we have the following cases:

- the node γ returned by Algorithm 1 is in fact either the exit node or the parex (because no false positives were found during the execution, or because by chance we anyway landed in the correct place): in this case everything goes smoothly as before;
- the returned node γ is not even an ancestor of $\mathrm{exit}(p)$: in this case, the first assertion (that $n_\gamma \preceq p$) fails;
- finally, if γ is a proper ancestor of $\mathrm{parex}(p)$, then Algorithm 2 proceeds as if γ was the parex but then the second assertion of the algorithm fails (γ' would in that case be an ancestor of the parex, and not the exit node as expected).

Thus, when executing Algorithm 2 in the signature-based case, we have to actually verify the assertions, which requires time $O(1 + (m \log \sigma)/w)$ and $O(1 + (m \log \sigma)/B)$ I/Os, as we have to compare the *whole name* n_γ with p (the cost of verifying the assertions covers also the cost of the comparisons with extents discussed in the proof of Theorem 6). If either assertion fails, we have to resort to the standard naive trie search to look for the exit node. The naive search requires, of course, time $O(m\sigma)$. Summing up:

▶ **Theorem 7.** *Under a 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 Algorithm 2 returns the correct result with probability at least $1 - o(1/n^c)$; otherwise, it detects an assertion error, in which case it returns the correct result by resorting to the standard naive search on the trie, which requires $O(m\sigma)$ time and I/Os.*

4 Suffix trees and suffix arrays

For a given string $s \in \Sigma^*$ of length $|s| = n$, let $\mathrm{Suff}(s\$)$ be the set of all the nonempty suffixes of $s\$$. We write $T(s)$ as an abbreviation for $T(\mathrm{Suff}(s\$))$. The trie $T(s)$ is called the *suffix tree* of s. As an example, in Figure 2 we show $T(\mathrm{ABRACADABRA})$.

Observe that $T(s)$ is a trie over the alphabet $\hat{\Sigma}$ (the addition of \$ is needed to make the language of suffixes prefix-free). It is worth noticing that in most papers on suffix trees there are no labels on nodes; instead, arcs are labelled with a nonempty strings (and the arcs to the children of a node have strings that differ in the first character). Our trees can be transformed into this (perhaps more standard) representation by removing all node labels and changing the label of each arc (α, α') to $c_{\alpha,\alpha'} c_{\alpha'}$. For the sake of comparison, we show in Figure 3 the alternative representation of the same trie of Figure 2.

Although in theory one could build the suffix tree of a string *explicitly*, much more (space- and time-) efficient approaches are available, based on suffix arrays. The *suffix array* [11] sa of s is the permutation of $\{0, 1, \ldots, n\}$ such that

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

Let us write s_i as an abbreviation of $s\$[\mathrm{sa}[i]\,..]$. It is also convenient to define the *lcp array* lcp defined by letting $\mathrm{lcp}[0] = \mathrm{lcp}[n+2] = 0$ and $\mathrm{lcp}[i+1]$, $0 \leq i \leq n$, as the length of the longest common prefix between s_i and s_{i+1}. In Figure 4, we show the suffix array and the lcp array for our running example $s = \mathrm{ABRACADABRA}$.

Every node α in the suffix tree can be identified with the interval $[\ell_\alpha \,..\, r_\alpha]$ of indices such that $k \in [\ell_\alpha \,..\, r_\alpha]$ if and only if s_k is the extent of one of the leaves that are descendants of α. The interval $[\ell_\alpha \,..\, r_\alpha]$ is called the *lcp-interval* of node α [1]. It is easy to see that:
- the lcp-interval of the root is $[0 \,..\, n]$;
- leaves are the only nodes whose lcp-interval is a singleton;

Z

ε	$\to$	$\text{root} = [0\,..\,11]$
A	$\to$	$\alpha_0 = [0\,..\,4]$
BR	$\to$	$\alpha_1 = [5\,..\,6]$
RA	$\to$	$\alpha_2 = [9\,..\,10]$
ABRA	$\to$	$\alpha_3 = [0\,..\,1]$

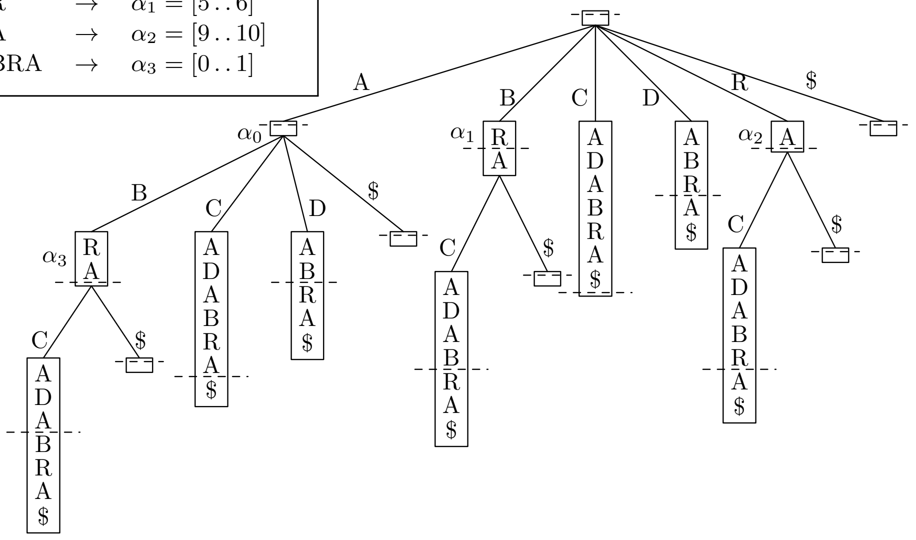

Figure 2 The zuffix tree T(ABRACADABRA) and its z-map (with lcp-intervals).

- the lcp-intervals of the children α_0, α_1, …, α_k of a node α are an ordered partition of the lcp-interval of their parent, where the ordering is established by the lexicographic order of the characters c_{α,α_i}.

Not only there are efficient algorithms to build the suffix array (and the associated lcp array) [10, 7, 12]: at the price of one additional (and very compressible) array 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]. More precisely, we can enumerate the lcp-intervals $[\ell\,..\,\ell_1 - 1]$, $[\ell_1\,..\,\ell_2 - 1]$, …, $[\ell_k\,..\,r]$ corresponding to the children of the node whose lcp-interval is $[\ell\,..\,r]$. Each child is enumerated in constant time.

It is easy to check that given a node with lcp-interval $[\ell\,..\,r]$ we can compute all the data we need (name, extent, etc.) using only the suffix and the lcp arrays:

$$\begin{aligned}
\left|n_{[\ell,r]}\right| &= \begin{cases} 0 & \text{if } \ell = 0 \text{ and } r = n \text{ (root)} \\ 1 + \max(\mathrm{lcp}[\ell], \mathrm{lcp}[r+1]) & \text{otherwise} \end{cases} \\
\left|e_{[\ell,r]}\right| &= \begin{cases} n - \mathrm{sa}[\ell] + 1 & \text{if } \ell = r \text{ (leaf)} \\ \mathrm{lcp}[a] & \text{otherwise} \end{cases} \\
\left|h_{[\ell,r]}\right| &= \text{the 2-fattest number in } \left[\left|n_{\ell,r}\right|\,..\,\left|e_{\ell,r}\right|\right] \\
n_{[\ell,r]} &= s\left[\mathrm{sa}[\ell]\,..\,\mathrm{sa}[\ell] + \left|n_{[\ell,r]}\right| - 1\right] \\
e_{[\ell,r]} &= s\left[\mathrm{sa}[\ell]\,..\,\mathrm{sa}[\ell] + \left|e_{[\ell,r]}\right| - 1\right]
\end{aligned}$$

where a is the left extreme of the lcp-interval of the second child of $[\ell\,..\,r]$.

Getting back again to the example depicted in Figure 2, consider the leftmost grandchild of the root, corresponding to the lcp-interval $[0\,..\,1]$; its children are both leaves and correspond to $[0]$ and $[1]$. Its name has length $1 + \max(\mathrm{lcp}[0], \mathrm{lcp}[2]) = 2$ (see Figure 4) and its extent has length $\mathrm{lcp}[1] = 4$. Since $\mathrm{sa}[0] = 0$, name and extent are $s[0\,..\,1] = AB$ and $s[0\,..\,3] = ABRA$, respectively. Its first child $[0]$ has name of length $1 + \max(\mathrm{lcp}[0], \mathrm{lcp}[1]) = 5$ (its name is in fact ABRAC), and extent of length $n - \mathrm{sa}[0] + 1 = 12$ (which is in fact ABRACADABRA\$).

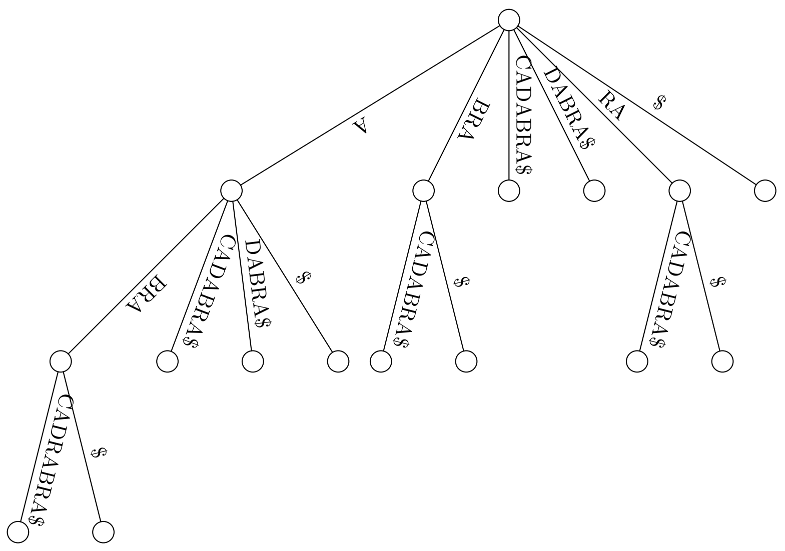

Figure 3 The suffix tree T(ABRACADABRA) in the alternative representation used in most papers about suffix trees.

5 Zuffification

As we explained, there are very efficient (in fact, even very well engineered) algorithms to build a suffix tree or an (enhanced) suffix array in linear time. What we want to do is adding to them a z-map gadget (in the case of suffix arrays, the z-map is that of the suffix tree that the suffix array implicitly represents). We call this process *zuffification* (which looks really nice for a spell), and resulting data structures are called the *zuffix tree* and the *(enhanced) zuffix array.*

The idea is very simple: we perform a depth-first visit of the suffix tree. For each internal node, we can compute its handle (as explained in the previous section), the corresponding hash, and store the correspondence between the hash and the node. We then check for collisions in the hash values of handles, possibly restarting with a different hash function if some collision is found, and finally build a static constant-time dictionary mapping the hashes to the nodes.

Of course, computing a hash for x needs time $O(1 + (|x| \log \sigma)/w)$. However, we can make hashing time constant by resorting to a particular kind of *rolling hashing*: in particular, we want the two following properties to be true:

1. Given a string $x \in \Sigma^*$ and $c \in \Sigma$, we have $H(xc) = f(H(x), c)$, where f can be computed in constant time.
2. Given strings $x, y \in \Sigma^*$, we have $H(y) = g(H(xy), H(x))$, where g can be computed in constant time.

If H is chosen in this way, we can build in linear time (by the first property) a table recording the hashes of the prefixes of the text s. At that point, computing the hash of a(ny) substring of s requires constant time by the second property (the values of $H(x)$ and $H(y)$ being found in the table). Several types of hash functions have the properties described above: the list includes hashing based on cyclic polynomials [15], Karp–Rabin hashing [8] and hashing based on the remainder of the division by a general irreducible polynomial [14] (the string is mapped to a string of bits and then interpreted as a polynomial over $\mathbf{F}_2$). Thus,

i	$\mathrm{sa}[i]$	$\mathrm{lcp}[i]$	$s_i = s\$[\mathrm{sa}[i]\,..]$
0	0	0	ABRACADABRA$
1	7	4	ABRA$
2	3	1	ACADABRA$
3	5	1	ADABRA$
4	10	1	A$
5	1	0	BRACADABRA$
6	8	3	BRA$
7	4	0	CADABRA$
8	6	0	DABRA$
9	2	0	RACADABRA$
10	9	2	RA$
11	11	0	$
12		0	

Figure 4 The suffix array and lcp array for the string ABRACADABRA.

▶ **Theorem 8.** *Zuffification can be performed in expected linear time (in fact, with high probability) both for suffix trees and suffix arrays.*

A practical improvement that slows down the construction time, but reduces significantly the space usage, is that of recording the hashes only for prefixes whose length is multiple of some value $O(w)$. In the cases above, it possible to still compute the hash of a substring quickly, albeit in some case $O(w)$ operations might be necessary (this depends on which instructions are considered to be atomic: for example, modern processor have constant-time multiplication of polynomial on $\mathbf{F}_2$). However, we have to store much fewer prefix hashes.

5.1 Searching with zuffix arrays

We can finally completely rebuild the spell of the fat sorceress. Given a string s, we build in linear time its enhanced suffix array, and simulate in linear time a visit of the associated suffix tree to build the z-map, thus obtaining the zuffix array. The latter operation works in expected linear time, because there might be collisions.

To search the zuffix array, we apply Theorem 7 to our setting, obtaining the same bounds. More importantly, since the nodes of the simulated suffix tree are exactly the intervals of the suffix array containing the starting points of the suffixes we are looking for, at the end of Algorithm 2 we have found all the locations of the search pattern:

▶ **Theorem 9.** *Under a full randomness assumption, given a pattern p, a zuffix array (in time $O((m\log\sigma)/w + \log m + \sigma)$ and with $O((m\log\sigma)/B + \log m + \sigma)$ I/Os) returns the interval of the underlying suffix array containing the positions at which p appears, with probability at least $1 - o(1/n^c)$; otherwise, it returns the same result in time $O(m\sigma)$.*

Significant practical improvements to the space used by the z-map can be obtained using the following theorem, which can be proved by adapting the proof of Theorem 5:

▶ **Theorem 10.** *Let L be a prefix-free language, and S a parent-closed set of nodes of $T(L)$. Consider a map Z sending $h_\alpha \to \alpha$, $\alpha \in S$. Then, Algorithm 1 returns the lowest ancestor in S of the exit node of the pattern, or its parent.*

The theorem above opens the door to a number of interesting space-time tradeoffs: for example, eliminating c levels of leaves in Definition 4 one obtains a much smaller map, but now locating the actual exit node (Algorithm 2) may require to go down c levels in the trie, using time $O((m \log \sigma)/w + \log m + c\sigma)$ and with $O(m \log \sigma / B + \log m + c\sigma)$ I/Os.

An even more interesting application is in DNA searching: usually the databases are very large, but the patterns are several orders of magnitude shorter: by building a z-map containing only handles shorter than ℓ, the search for patterns shorter than ℓ will not be affected, but the map will be much smaller (longer patterns will still be searchable, possibly using time $O(m\sigma)$).

6 Conclusions

The fat sorceress was actually quite good at spells: given a pattern p, after a search that is logarithmic in $|p|$, a zuffix array tests p against s scanning from at most σ positions. In practice, this means at most σ cache misses in the test phase (this fact is only partially expressed by the I/Os in the cache-oblivious model, as the model does not consider prefetching). The result needs an uncompressed original text, but in the context of the rise of modern low-cost fast storage, like solid-state drives, this limitation does not seem so serious. We remark that in our description we used enhanced suffix arrays, but nothing prevents zuffification of compressed suffix trees [16], compressed suffix arrays [6] or even of the FM-index [4].

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of 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 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, jul 2005.

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

6 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.

7 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

8 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

9 Donald E. Knuth. *The Art of Computer Programming.* Addison–Wesley, 1973.

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

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

12 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE, 2009.

13 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.

14 W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

15 Eugene Prange. Cyclic error-correcting codes in two symbols. Technical note AFCRC-TN-57-103, Air Force Cambridge Research Center, 1957.

16 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.