Alphabet-Independent Compressed Text Indexing

DJAMAL BELAZZOUGUI
Université Paris Diderot
GONZALO NAVARRO
University of Chile

Self-indexes are able to represent a text within asymptotically the information-theoretic lower bound under the kth order entropy model, and offer access to any text substring and indexed pattern searches. Their time complexities are not optimal, however; in particular they are always multiplied by a factor that depends on the alphabet size. In this paper we achieve, for the first time, *full alphabet-independence* in the time complexities of self-indexes, while retaining space optimality. We obtain also some relevant byproducts.

Categories and Subject Descriptors: E.1 [**Data Structures**]: ; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

General Terms: Algorithms

Additional Key Words and Phrases: Compression, Text Indexing, Succinct Data Structures, Suffix Trees.

1. INTRODUCTION

Text indexes are data structures built on a text $T[1,n]$, over alphabet $[1,\sigma]$, that support pattern searches. The most popular text indexes in stringology are the suffix tree [Weiner 1973; McCreight 1976; Apostolico 1985; Ukkonen 1995] and the suffix array [Gonnet et al. 1992; Manber and Myers 1993; Kärkkäinen and Sanders 2003]. The importance of these data structures cannot be overemphasized: one can find whole books dedicated to problems that are solved with them in areas like Information Retrieval [Baeza-Yates and Ribeiro-Neto 2011], Bioinformatics [Gusfield 1997], and Stringology [Crochemore and Rytter 2003].

Among many other more sophisticated functionalities, suffix trees can *count* the occurrences of a pattern $P[1,m]$ in T in time $t_{\mathsf{count}} = O(m)$, and even $t_{\mathsf{count}} = O(m/\log_\sigma n)$. Suffix arrays achieve time $t_{\mathsf{count}} = O(m \log n)$, and even $t_{\mathsf{count}} = O(m + \log n)$. After counting, they can *locate* the position of any such occurrence in T in time $t_{\mathsf{locate}} = O(1)$. As the text is available, one can *extract* any text substring $T[i, i+\ell-1]$ in optimal time $t_{\mathsf{extract}} = O(\ell/\log_\sigma n)$.

However, the $O(n \log n)$-bit space complexity of both suffix trees and suffix arrays, given that representing the plain text needs only $n \log \sigma$ bits, renders these classical

First author partially supported by the French ANR-2010-COSI-004 MAPPI Project. Both authors partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.
An early partial version of this paper appeared in *Proc. ESA'11*.
Authors' address: Djamal Belazzougui, Dept. of Computer Science, University of Helsinki, Finland, `djamal.belazzougui@gmail.com`. Gonzalo Navarro, Dept. of Computer Science, University of Chile, `gnavarro@dcc.uchile.cl`

structures unapplicable for large text collections, precisely in the cases where they would be most useful.

Compressed text *self-indexes* [Navarro and Mäkinen 2007] are an answer to the space problem. They represent a text T within asymptotically the space needed to store it in compressed form. Within this compressed space, self-indexes allow not only extracting any substring of T, but also counting and locating the occurrences of patterns. They have been shown to be practical and surprisingly competitive [Ferragina et al. 2009].

A popular model to measure text compressibility is the *empirical entropy* [Manzini 2001]: the kth order empirical entropy of T, $0 \leq H_k(T) \leq H_{k-1}(T) \leq \log \sigma$,[1] is a lower bound to the bits per symbol emitted by any statistical compressor that models T considering the context of k symbols that precede the one to encode.

Starting with the FM-index [Ferragina and Manzini 2000] and the Compressed Suffix Array (CSA) [Grossi and Vitter 2000; Sadakane 2000], self-indexes have evolved up to a point where they have reached asymptotically optimal space within the kth order entropy model, that is, $nH_k(T)+o(n \log \sigma)$ bits [Sadakane 2003; Grossi et al. 2003; Ferragina and Manzini 2005; Ferragina et al. 2007; Barbay et al. 2007; Barbay et al. 2010; Navarro and Mäkinen 2007]. This remarkable space reduction achieved by self-indexes is paid in terms of higher time complexities compared to the classical suffix trees and arrays.

Table I lists the current space-optimal self-indexes. All follow a model where a sampling step s is chosen (which costs $O((n \log n)/s)$ bits, so we need $s = \omega(\log_\sigma n)$ for asymptotic space optimality), and then locating an occurrence costs s multiplied by some factor that depends on the alphabet size σ. The time for extracting is linear in $s + \ell$, and is also multiplied by the same factor. Recently, the concept of asymptotic space optimality has been carried out one step further, achieving $o(nH_k(T)) + o(n) \subseteq o(n \log \sigma)$ extra space [Barbay et al. 2010]. This is important for highly compressible texts.

The only index achieving locating and extracting times independent of σ is by Sadakane [2003], yet its counting time is the worst. On the other hand, various FM-indexes [Ferragina et al. 2007; Barbay et al. 2010] achieve $O(m)$ counting, $O(s)$ locating, and $O(s + \ell)$ extraction time when the alphabet is polylogarithmic in the text size, $\sigma = O(\text{polylog}(n))$. Achieving the same for larger alphabets, where $\log \sigma = \omega(\log \log n)$, has been open.[2]

Grossi et al. [2003] have described other structures that escape from this general scheme, however they need to use more than the optimal space. By using $(2 + \varepsilon)nH_k(T) + o(n \log \sigma)$ bits, for any $\varepsilon > 0$, they achieve the optimal $O(m/ \log_\sigma n)$ counting time, albeit with an additive polylogarithmic penalty of $p(n) = O(\log_\sigma^{(3+\varepsilon)/(1+\varepsilon)} n \log^2 \sigma)$. They can also achieve sublogarithmic locating time, $O(\log^{1/(1+\varepsilon)} n)$. Finally the extraction time is also optimal plus the polylogarithmic penalty, $O(\ell/ \log_\sigma n + p(n))$.

[1]Our logarithms are base 2.

[2]Since we will be interested in the case $\sigma = \omega(1)$, we can write space complexities of the form $o(n \log \sigma)$ without ambiguity on whether the sublinearity is on n or σ. Nevertheless, we will be more specific whenever possible. In particular, the notation $o(nH_k(T))$ will mean $o(n) \cdot H_k(T)$.

Source	Space ($+O(\frac{n \log n}{s})$)	Counting	Locating	Extracting
GGV03	$nH_k + o(n \log \sigma)$	$O(m \log \sigma + \log^4 n)$	$O(s \log \sigma)$	$O((s+\ell) \log \sigma)$
Sad03	$nH_k + o(n \log \sigma)$	$O(m \log n)$	$O(s)$	$O(s+\ell)$
FMMN07	$nH_k + o(n \log \sigma)$	$O(m \frac{\log \sigma}{\log \log n})$	$O(s \frac{\log \sigma}{\log \log n})$	$O((s+\ell) \frac{\log \sigma}{\log \log n})$
BHMR07	$nH_k + o(n \log \sigma)$	$O(m \log \log \sigma)$	$O(s \log \log \sigma)$	$O((s+\ell) \log \log \sigma)$
BGNN10	$nH_k + o(nH_k) + o(n)$	$O(m \frac{\log \sigma}{\log \log n})$	$O(s \frac{\log \sigma}{\log \log n})$	$O((s+\ell) \frac{\log \sigma}{\log \log n})$
BGNN10	$nH_k + o(nH_k) + o(n)$	$O(m \log \log \sigma)$	$O(s \log \log \sigma)$	$O((s+\ell) \log \log \sigma)$
Ours	$nH_k + o(nH_k) + O(n)$	$O(m)$	$O(s)$	$O(s+\ell)$

GGV03 = [Grossi et al. 2003]; Sad03 = [Sadakane 2003]; FMMN07 = [Ferragina et al. 2007]; BHMR07 = [Barbay et al. 2007]; BGNN10 = [Barbay et al. 2010].

Table I. Current and our new complexities for asymptotically optimal-space self-indexes. The space results (in bits) hold for any $k \leq \alpha \log_\sigma(n) - 1$ and constant $0 < \alpha < 1$, and any sampling parameter s. The counting time is for a pattern of length m and the extracting time for ℓ consecutive symbols of T. The space for Sadakane's structure [Sadakane 2003] refers to a more recent analysis [Navarro and Mäkinen 2007]; see also the clarifications in `www.dcc.uchile.cl/gnavarro/fixes/acmcs06.html`.

The main contribution of this article is the last row of Table I. We achieve, for the first time, *full alphabet independence* for all alphabet sizes, both in time and space complexity and within the strictest model of asymptotic space-optimality, at the sole price of converting an $o(n)$-bit redundancy into $O(n)$. This is an important step towards leveraging the time penalties incurred by asymptotically space-optimal compressed text indexes.

Our general strategy is to build on top of FM-indexes and CSAs, in particular using recent variants that support operation *select* in constant time (see Section 2.2 for the meaning of operations *select*, *rank*, and *access* on sequences) in order to achieve fast *locate* and *extract*. Achieving $O(m)$ time for *count* is much more difficult, because it builds on operation *rank*, which cannot be supported in constant time [Belazzougui and Navarro 2012]. To circumvent the use of this operation we use a combination of Compressed Suffix Trees (CSTs) and monotone minimal perfect hash functions (mmphfs). After we review in Section 2 some basic concepts on text indexing, the layout of the article is as follows.

(1) In Section 3 we review mmphfs and show, as an easy and immediate application to self-indexes, a way to achieve $O(s)$ locating time and $O(s+\ell)$ extracting time. This result is, however, incompatible with the rest of the article, because it needs constant-time *access* and for the rest we need constant-time *select*, and one cannot have both with $o(n \log \sigma)$ space redundancy, at least for $\log \sigma = \Theta(\log n)$ [Golynski 2009]. Thus we need a completely different approach in order to obtain also $O(m)$ counting time (however, see point (7)).

(2) In Section 4 we show how different aspects of the FM-index and CSA representations can be combined to obtain the same result of point (1), now assuming a representation giving constant-time *select*. This does not use mmphfs, and is a direct improvement on previous results [Barbay et al. 2010]. It involves extending operation *select* to operate in constant time on a string that has been partitioned to achieve kth order entropy [Ferragina et al. 2005].

(3) In Section 5 we give our first result towards $O(m)$-time counting, which has independent interest. We show how to enhance the CST of Sadakane [2007]

in order to speed up the important operation $child(v,c)$, which gives the child of a node v by a symbol c. In the original formulation [Sadakane 2007] this operation takes time $O(t_{\mathsf{SA}} \log \sigma)$, where t_{SA} is the time to compute the content of a cell of the suffix array or of its inverse permutation using the underlying FM-index or CSA (usually $t_{\mathsf{SA}} = t_{\mathsf{locate}}$). Using mmphfs, we improve the time to $O(t_{\mathsf{SA}})$ at the cost of $O(n \log\log \sigma)$ bits of space. This extra space is within the usual $o(n \log \sigma)$ bits of redundancy posed by CSTs. No other CST had achieved $O(t_{\mathsf{SA}})$ time [Fischer et al. 2009; Fischer 2010; Russo et al. 2011]. Only Sadakane [2007] had a way to achieve this time complexity, but this required much more space, $O(n \log \sigma)$ bits, which is not anymore compressed.

(4) In Section 6 we build on the result of point (3) to design a method that descends in a CST from the root towards the node representing a string of length m, in time $O(m)$, using $O(n \log t_{\mathsf{SA}})$ extra bits and assuming the underlying text index is the one described in point (2). Combined with that text index, we almost achieve our main goal, except that we add a redundancy of $O(n \log\log n)$. The main difficulty is how to represent the suffix tree skips succinctly, and the result also has independent interest for traversing CSTs in general.

(5) In Section 7 we finally achieve our goal. For this sake we do not use the suffix tree traversal considered in points (3) and (4), but instead simulate *backward search*, the native pattern matching technique of the FM-index, using mmphf-supported Weiner links [Weiner 1973] on the CST. This allows us doing the backward steps with operation *select* instead of *rank* and achieve $O(m)$ time for the whole process. We design a way to store the mmphfs so that the redundancy they pose is also compressed, $o(nH_k(T)) + O(n)$ bits. That is, we increase the best known redundancy by only $O(n)$ bits.

(6) In Section 8 we give the index of point (5) a stronger functionality of independent interest. This index carries out m steps in $O(m)$ time, but an individual backward step can take up to $O(m)$ time. In some applications it might be useful to support an individual backward step in constant time, for example for approximate matching on FM-indexes [Lam et al. 2008; Mäkinen et al. 2010], or in general to simulate any suffix tree traversal [Baeza-Yates and Gonnet 1996; Gusfield 1997] on an FM-index, or for other totally different tasks [Ohlebusch et al. 2010; Beller et al. 2011]. By adding more data to the index we achieve constant-time backward steps, yet the extra redundancy goes back to $O(n \log\log \sigma)$ bits.

(7) Finally, in Section 9 we close the circle by showing how one can achieve counting time $O(m+s)$ over the simple index proposed in point (1). For this sake we adapt the ideas developed in point (6).

2. COMPRESSED TEXT SELF-INDEXES

2.1 Suffix Tries, Trees, and Arrays

Given a text $T[1,n]$ over alphabet $[1,\sigma]$, each substring $T[i,n]$ is called a *suffix*. A *suffix trie* is a digital tree where all the suffixes of T have been inserted. More precisely, once the prefix of a suffix $T[i,n]$ is unique, the suffix trie node becomes a leaf recording the suffix position i. Thus the suffix trie has exactly n leaves, one per suffix of T. A *suffix tree* compacts the unary paths in the suffix trie, so that edges are labeled by strings. Note the strings labeling the children of a node start with distinct symbols. After compaction, only the $O(n)$ branching nodes of the suffix trie become *explicit* nodes in the suffix tree. The other suffix trie nodes, which are not represented in the suffix tree, are called *implicit* (implicit nodes can be thought of as hidden inside suffix tree edges). The suffix tree has $O(n)$ nodes: n leaves and at most $n-1$ internal nodes.

We assume that suffix tries and trees store the children of each node in lexicographic order of the (symbol or string) labels of the corresponding edges. If we collect their leaves in order, the result is the *suffix array* $A[1,n]$ of T. This is just a permutation of $[1,n]$ that sorts the suffixes of T in lexicographic order: $T[A[i],n] < T[A[i+1],n]$ for all i.

Suffix tries can be used for counting the occurrences of $P[1,m]$ in time $O(m)$ as follows: descend from the root according to the successive symbols $P[i]$. Constant time per node is achieved with a perfect hash function that stores the symbols labeling the edges toward the children of each node. If at some point the desired child does not exist, then P does not occur in T. Otherwise we arrive at a node v, and the number of leaves below v is the number of occurrences of P in T. The process on the suffix tree is similar, except that the edges are labeled by strings and thus we may need to consume several symbols of P to descend to a child. Actually the search may end at the middle of an edge leading to a node v, in which case the subtree rooted at v contains all the occurrences of P. The occurrences can be located by traversing all the leaves below v. On a suffix array we simply binary search the interval $A[sp,ep]$ containing all the suffixes that start with P. This takes $O(m \log n)$ time, or $O(m + \log n)$ with extra longest common prefix information.

For more complex traversals, the concept of suffix link and Weiner link are relevant. We say that a suffix trie or tree node *represents* the string that is obtained by concatenating the edge labels from the root to the node (the length of this string is called the *string depth* of the node). A *suffix link* in a suffix trie or tree is a pointer from a node u representing a string cX to the node v representing string X, $v = slink(u)$, where c is a symbol and X is a string. A kind of opposite function is called a *Weiner link*, $u = wlink(v,c)$. On the suffix tree, these functions are defined only on explicit nodes. Note that, if u is an explicit node, then $v = slink(u)$ is also explicit; similarly, if $wlink(v,c)$ is explicit, then v is also explicit.

Suffix trees and arrays take $O(n \log n)$ bits of space. While the first constructions required $O(n \log \sigma)$ time for suffix trees [Weiner 1973; McCreight 1976; Ukkonen 1995] and $O(n \log n)$ time for suffix arrays [Manber and Myers 1993], $O(n)$ time construction, independently of the alphabet size, was achieved later for both [Farach 1997; Kärkkäinen and Sanders 2003]. For more information please refer to the abundant bibliography on suffix trees and arrays [Apostolico 1985; Gusfield 1997; Crochemore and Rytter 2003; Baeza-Yates and Ribeiro-Neto 2011].

2.2 Compressed Sequence Representations and Zero-Order Entropy

An important subproblem that arises in compressed text self-indexing is that of representing a sequence $S[1,n]$, over an alphabet $[1,\sigma]$, supporting the following operations:

—$access(S,i) = S[i]$, in time t_{access};

—$rank_c(S,i)$, the number of times symbol c appears in $S[1,i]$, in time t_{rank}; and

—$select_c(S,i)$, the position in S of the ith occurrence of c, in time t_{select}.

For the particular case of bitmaps (i.e., $\sigma = 2$, where the symbols are called 0 and 1), constant-time operations can be achieved using $n + o(n)$ bits [Jacobson 1989; Clark 1996; Munro 1996]. When the number of 1s in S, n_1, is much smaller than n, a compressed representation [Raman et al. 2002] taking $\log \binom{n}{n_1} + o(n) = n_1 \log \frac{n}{n_1} + O(n_1) + o(n)$ bits and still solving the operations in constant time, is convenient. The space can also be written as $nH_0(S) + o(n)$ bits, where $H_0(S)$ is the *zero-order entropy* of S. In general,

$$H_0(S) \quad = \quad \sum_{c \in [1,\sigma]} \frac{n_c}{n} \log \frac{n}{n_c} \quad \le \quad \log \sigma, \tag{1}$$

where n_c is the number of times c occurs in S.

General sequences can also be represented within asympotically zero-order entropy space, $nH_0(S) + o(n \log \sigma)$ bits, and even $nH_0(S) + o(nH_0(S)) + o(n)$. Among the many compressed sequence representations [Golynski et al. 2006; Ferragina et al. 2007; Barbay et al. 2007; Barbay et al. 2010; Grossi et al. 2010], we emphasize two results for this article. The first corresponds to Theorem 1, variant (i), of the recent result of Barbay et al. [2010]. The second is obtained by using the same theorem, yet replacing the representation of Golynski et al. [2006] for the sequences of similar frequency, by another recent result of Grossi et al. [2010] (the scheme itself compresses to $H_k(S) + o(n \log \sigma)$ bits, with stricter restrictions on k; when combining with Barbay et al. we only use the fact that it takes $n \log \sigma + o(n \log \sigma)$ bits).

Lemma 1. [Barbay et al. 2010; Grossi et al. 2010] *A sequence $S[1,n]$ over alphabet $[1,\sigma]$ can be represented within $nH_0(S) + o(n(H_0(S)+1)) + O(\sigma \log n)$ bits of space, so that the operations are supported in times either (1) $t_{\mathsf{access}} = t_{\mathsf{rank}} = O(\log\log\sigma)$ and $t_{\mathsf{select}} = O(1)$, or (2) $t_{\mathsf{select}} = t_{\mathsf{rank}} = O(\log\log\sigma)$ and $t_{\mathsf{access}} = O(1)$.*

2.3 Compressed Suffix Arrays (CSAs)

Compressed Suffix Arrays (CSAs) [Grossi and Vitter 2000; Sadakane 2003; Grossi et al. 2003] are self-indexes representing $T[1,n]$ and its suffix array $A[1,n]$ in compressed form. Their main component is function $\Psi(i) = A^{-1}[(A[i] \mod n)+1]$, which lets us advance virtually in T: If $A[i] = j$, then $A[\Psi(i)] = j+1$, that is, $\Psi(i)$ tells where in A is the pointer to the text position next to that pointed from $A[i]$. CSAs represent array Ψ within compressed space and can provide constant-time access time to any value $\Psi(i)$. A sparse bitmap $D[1,n]$ marks position 1 and the positions i such that $T[A[i]] \neq T[A[i-1]]$, that is, the points in A where the first symbol of the pointed suffixes changes. In addition, the distinct symbols in T are stored in a string $Q[1,\sigma]$, in lexicographic order. By storing D in compressed form [Raman et al. 2002], D and Q occupy $O(\sigma \log n) + o(n)$ bits and we have constant-time *rank* and *select* on D. Then we can compute any $T[A[i]] = Q[rank_1(D,i)]$ in $O(1)$ time. Moreover, the relation $T[A[i]+k] = T[A[\Psi^k(i)]]$ lets us retrieve any string $T[A[i], A[i]+\ell-1]$ in time $O(\ell)$.

This enables the simple binary search for $P[1,m]$ on A, as we can extract in $O(m)$ time any string $T[A[i],A[i]+m-1]$ needed for comparing P with the suffix of T pointed from any $A[i]$. This allows us to *count* in time $O(m \log n)$.

For locating those occurrences, we sample T regularly every s positions. We store a bitmap $V[1,n]$, where $V[j]=1$ iff $A[j] \mod s = 0$, plus an array $S_A[rank_1(V,j)] = A[j]/s$ for those j where $V[j]=1$. To find $A[i]$ we compute successively $j=\Psi^k(i)$ for $k=0,1,\ldots,s-1$ until $V[j]=1$; then $A[i]=S_A[rank_1(V,j)]\cdot s+k$. Thus we can *locate* any occurrence in time $O(s)$, using $O((n\log n)/s)$ extra bits.

For extracting arbitrary substrings $T[i,i+\ell-1]$ we use basically the same sampling. We store $S_T[j] = A^{-1}[1+s\cdot j]$ for $j=0,1,\ldots,n/s$. Then we compute $j=\lfloor (i-1)/s \rfloor$ and extract the longer substring $T[j\cdot s+1, i+\ell-1]$. Since the extraction starts from $A[S_T[j]]$ we obtain the first character as $c = T[A[S_T[j]]] = rank_1(D,S_T[j])$, and we use Ψ to find the positions in A pointing to the consecutive characters to extract. Thus *extract* requires time $O(s+\ell)$, again using $O((n\log n)/s)$ extra bits.

2.4 The Burrows-Wheeler Transform and High-Order Entropy

The Burrows-Wheeler Transform (BWT) [Burrows and Wheeler 1994] of a text $T[1,n]$ is a string $T^{bwt}[1,n]$, where the symbols of T have been reordered. It is defined as $T^{bwt}[i] = T[A[i]-1]$ (or $T[n]$ if $A[i]=1$). Thus there is a close relation between T^{bwt} and A: T^{bwt} is the sequence of symbols preceding the suffixes of T once they are lexicographically sorted.

The BWT is a reversible transform. Assuming T finishes with a special character \$ smaller than all the others, it holds $A[1]=n$ and thus $T[n]=T^{bwt}[1]$. Now, given that we have decoded $T[i]=T^{bwt}[j]$, it holds $T[i-1]=T^{bwt}[LF(j)]$, where $LF(j) = C[c] + rank_c(T^{bwt},j)$, $c = T^{bwt}[j]$, and C is a small array storing in $C[c]$ the number of symbols $< c$ in T. Note this function LF is the inverse of Ψ: If $A[j]=i+1$, then $A[LF(j)]=i$.

The BWT makes the text more compressible, in the following sense. The *kth order entropy* of T, $H_k(T)$, is a lower bound on the number of bits per symbol emitted by a statistical compressor that encodes each symbol of T as a function of the next k symbols.[3] It holds $0 \le H_k(T) \le H_{k-1}(T) \le H_0(T) \le \log\sigma$ for any $k>0$. The precise formula is

$$\begin{aligned} H_k(T) &= \sum_{W\in[1,\sigma]^k} \frac{n(T_W)}{n} H_0(T_W) &&= \sum_{W\in[1,\sigma]^k} \frac{n(T_W)}{n} \sum_{c\in[1,\sigma]} \frac{n_c(T_W)}{n(T_W)} \log\frac{n(T_W)}{n_c(T_W)} \\ &= \sum_{cW\in[1,\sigma]^{k+1}} \frac{n_c(T_W)}{n}\log\frac{n(T_W)}{n_c(T_W)} &&= \sum_{cW\in[1,\sigma]^{k+1}} \frac{n(T_{cW})}{n}\log\frac{n(T_W)}{n(T_{cW})}, \quad (2)\end{aligned}$$

where T_X is a string formed by the characters preceding string X in T, $n(T_X) = |T_X|$, and $n_c(T_X)$ is the number of occurrences of symbol c in string T_X.

Now let us partition $S = T^{bwt}$ into σ^k *contexts* S_W, for all $W\in[1,\sigma]^k$, as follows. If $A[sp(W),ep(W)]$ is the range of A where the suffixes start with string W, then

[3] As compression proceeds left to right, using the preceding k symbols is much more popular, but the definition we use relates better with the BWT. The difference in entropy when using one or the other definition is negligible [Ferragina and Manzini 2005].

$S_W = S[sp(W), ep(W)]$. The remarkable property is that $T_W = S_W$, as these are the symbols that precede the occurrences of W in T. Thus, by compressing each context S_W of S to zero-order entropy, we obtain kth order entropy compression for the original string T.

2.5 The FM-index

The FM-index [Ferragina and Manzini 2005] is a compressed self-index built on the BWT. In its modern form [Ferragina et al. 2007], the index cuts T^{bwt} optimally into partitions, and represents each partition as a zero-order compressed sequence supporting *rank* and *access* operations. From their analysis [Ferragina et al. 2007] it follows that if each such sequence S is represented within $|S|H_0(S)+o(|S|H_0(S))+o(|S|)+O(\sigma \log n)$ bits of space, then the overall space of the index is $nH_k(T)+o(nH_k(T))+o(n)+O(\sigma^{k+1}\log n)$, for any k. The latter term is usually removed by assuming $k \le \alpha \log_\sigma(n)-1$ and constant $0<\alpha<1$. This is the space Barbay et al. [2010] achieve, and the best space reported so far for compressed text indexes under the kth order entropy model (see Table I).

Although the FM-index computes the optimal partitioning of T^{bwt} [Ferragina et al. 2005], we can analyze it as if it cut the partitions using contexts of length k in order to achieve kth order compression (and the space will always be at least that good [Ferragina et al. 2007]). Thus assume we cut the BWT into σ^k partitions. The FM-index marks them in a sparse bitmap E that is represented within $O(\sigma^k \log n)+o(n)$ bits and offers constant-time *rank* and *select* [Raman et al. 2002]. In order to compute function LF, we need to compute *access* and *rank* over the whole T^{bwt}, not only over a partition. Bitmap E lets us know which position of which partition must be accessed to solve *access* or *rank*. For the latter, we also need to know *rank* up to the beginning of each partition. This can be tabulated within $O(\sigma^{k+1}\log n)$ bits. Therefore the time to compute LF is $t_{\mathsf{LF}} = O(t_{\mathsf{access}}+t_{\mathsf{rank}})$, where t_{access} and t_{rank} refer to the times to execute the operations within a single partition.

The time to compute LF impacts all the times of the FM-index. We use LF for locating and extracting just like CSAs use Ψ, but traversing T in backward direction: by sampling T regularly every s positions, any cell $A[i]$ can be computed in time $O(s \cdot t_{\mathsf{LF}})$, and any substring of T of length ℓ can be extracted in time $O((s+\ell)\cdot t_{\mathsf{LF}})$. In order to achieve $t_{\mathsf{LF}} = O(1)$, we will have to avoid the use of *rank*, as it cannot be computed in constant time [Belazzougui and Navarro 2012].

The FM-index has a particular and more efficient way of counting, that is, determining the suffix array area $A[sp, ep]$ where pattern P occurs, so that its occurrences can be counted as $ep - sp + 1$ and each of those occurrence positions can be located using $A[i]$, for $sp \le i \le ep$. Counting is done via the so-called *backward search*, which processes the pattern in reverse order. Let $A[sp, ep]$ be the interval for $P[i+1, m]$, then the interval for $P[i, m]$ is $A[sp', ep']$, where $sp' = C[c] + rank_c(T^{bwt}, sp-1)+1$ and $ep' = C[c] + rank_c(T^{bwt}, ep)$, where $c = P[i]$. This requires computing $O(m)$ times operation *rank*, yet this *rank* operation is of a more general type than for LF (i.e., it does not hold $T^{bwt}[i] = c$ for $rank_c(T^{bwt}, i)$). Therefore, achieving linear time for counting will require a more elaborate technique to get rid of the *rank* operation.

2.6 Compressed Suffix Trees (CSTs)

CSAs and FM-indexes support operations *count*, *locate*, and *extract*, but they lack the richer functionality offered by suffix trees. A *compressed suffix tree (CST)* is a compact data structure offering those more sophisticated operations. A CST is usually built on top of a CSA or FM-index, which simulates a suffix array, and adds some extra space to support more operations. In fact the underlying index must support operations $A[i]$ as well as $A^{-1}[j]$, both in time called t_{SA} (in compressed text indexes it usually holds $t_{\mathsf{SA}} = t_{\mathsf{locate}}$; this holds in particular with the sampling scheme described in Section 2.3).

Sadakane [2007] proposed the first CST. It uses $O(n)$ bits on top of a CSA and supports many complex operations in constant time (e.g., from going to the parent, or following a suffix link, to computing lowest common ancestors). Part of the $O(n)$ bits are spent in $2n$ balanced parentheses that, via a DFS traversal, represent the topology of the suffix tree. That is, each suffix tree node corresponds to an opening parenthesis, and all its subtree is recursively represented before closing that parenthesis.

Surprisingly, one of the most basic suffix tree operations is rather slow. Operation $child(v, c)$, which returns the child of v by (a string starting with) symbol c, takes $O(t_{\mathsf{SA}} \log \sigma)$ time. Sadakane shows how to reduce this time to $O(t_{\mathsf{SA}})$, yet the price is huge: $O(n \log \sigma)$ further bits of space.

Other later CSTs [Fischer et al. 2009; Fischer 2010; Russo et al. 2011] aimed mostly at reducing the $O(n)$ bits of redundancy, but they are usually slower for all the operations. In particular, none of them achieve $O(t_{\mathsf{SA}})$ time for operation *child*.

3. MONOTONE MINIMAL PERFECT HASH FUNCTIONS, AND A WARM UP

A *monotone minimal perfect hash function (mmphf)* [Belazzougui et al. 2009a; 2009b] $f : [1, u] \rightarrow [1, n]$, for $n \le u$, assigns consecutive values $1, 2, \ldots, n$ to domain values $u_1 < u_2 < \ldots < u_n$, and arbitrary values to the rest. Seen another way, it maps the elements of a set $\{u_1, u_2, \ldots, u_n\} \subseteq [1, u]$ into consecutive values in $[1, n]$. Yet a third view is a bitmap $B[1, u]$ with n bits set; then $f(i) = rank_1(B, i)$ where $B[i] = 1$ and $f(i)$ is arbitrary where $B[i] = 0$.

A mmphf on B does not give sufficient information to reconstruct B, and thus it can be stored within less than $\log \binom{u}{n}$ bits, more precisely $O(n \log\log \frac{u}{n} + n)$ bits. This allows using it to speed up operations while adding an extra space that is asymptotically negligible. A second type of mmphf we will use later in the article requires $O(n \log\log\log \frac{u}{n} + n)$ bits of space, and answers queries in time $O(\log\log \frac{u}{n})$.

As a simple application of mmphfs, we show how to compute function LF on a sequence S within time $O(t_{\mathsf{access}})$, by using additional $O(|S|(\log H_0(S) + 1))$ bits of space. For each symbol c appearing in S we build a mmphf f_c which records all the positions at which c appears in S. This hash function occupies $O(n_c(\log\log \frac{|S|}{n_c} + 1))$ bits, where n_c is the number of occurrences of c in S. Summing up over all characters we get additional space usage $O(|S|(\log H_0(S) + 1))$ bits.

LEMMA 2. *If c occurs n_c times in S, then $\sum_{c \in [1,\sigma]} n_c \log\log \frac{|S|}{n_c} \le |S| \log H_0(S)$.*

PROOF. According to the log-sum inequality, given σ pairs of numbers $a_c, b_c > 0$, it holds $\sum a_c \log \frac{a_c}{b_c} \ge (\sum a_c) \log (\sum a_c / \sum b_c)$. Use $a_c = n_c/|S|$ and $b_c = -a_c \log a_c$

to obtain the claim, using Eq. (1). □

The LF function can now be easily computed in time $O(t_{\mathsf{access}})$ as $LF(i) = C[c] + f_c(i)$, where $c = T^{bwt}[i]$, since we know that f_c is well-defined at i. Therefore the time of the LF function becomes $O(1)$ if we have constant-time access to T^{bwt}. Consider now partitioning the BWT as in Section 2.4. Our extra space is $O(|S|(\log H_0(S) + 1))$ within each partition S of the BWT. This adds up to $O(n(\log H_k(T) + 1))$.

Lemma 3. *Let $T^{bwt} = S_1 S_2 \ldots$ Then $\sum_i |S_i| \log H_0(S_i) \leq |T| \log H_k(T)$.*

Proof. We apply again the log-sum inequality, using $a_i = |S_i|$ and $b_i = |S_i| H_0(S_i)$. Then Eq. (2) completes the proof. □

Therefore, we obtain the following result.

Lemma 4. *By adding $O(n(\log H_k(T) + 1))$ bits to an FM-index built on text $T[1, n]$ over alphabet $[1, \sigma]$, one can compute the LF function in time $t_{\mathsf{LF}} = O(t_{\mathsf{access}})$, where t_{access} is the time needed to access any element in T^{bwt}.*

If we choose the sequence representation (2) of Lemma 1, so that $t_{\mathsf{access}} = O(1)$, we achieve constant-time LF computation (Lemma 4) and, consequently, locate time $O(s)$ and extract time $O(s + \ell)$, at the cost of $O((n \log n)/s)$ extra bits.

The sequence representation for each partition S takes $|S|H_0(S) + o(|S|H_0(S)) + o(|S|) + O(\sigma \log n)$ bits. Added over all the partitions (recall Section 2.5), this gives the main space term $nH_k(T) + o(nH_k(T)) + o(n) + O(\sigma^{k+1} \log n)$, as explained. On top of this, Lemma 4 requires $O(n(\log H_k(T) + 1))$ bits. This is $o(nH_k(T)) + O(n)$ if $H_k(T) = \omega(1)$, and $O(n)$ otherwise.

To reduce the $O(n)$ space term to $o(n)$, we will show how to ensure that the space of the mmphf of each S is not only $O(|S|(\log H_0(S) + 1))$, but also $o(|S|H_0(S)) + o(|S|)$. Again, this clearly holds unless $H_0(S) = O(1)$, so we focus on this case.

Note that Barbay et al. [2010] (on which we are building in Lemma 1) achieves constant time for all the operations when the frequency of the involved symbol exceeds $|S|/\log |S|$.[4] Therefore, we only need to build mmphfs f_c for those symbols c with frequency $n_c \leq |S|/\log |S|$. Note that such a symbol contributes $\log \frac{|S|}{n_c} \geq \log\log |S|$ bits to $H_0(S)$. Therefore, if $H_0(S) = O(1)$, then the total number of occurrences of those less frequent symbols must be $O(|S|/\log\log |S|)$.

We further divide those symbols into medium frequency, that is, $|S|/2^{(\log\log |S|)^2} \leq n_c \leq |S|/\log |S|$, and low frequency, $n_c < |S|/2^{(\log\log |S|)^2}$. From the medium frequency symbols, the space per occurrence spent by the mmphf is $O(\log\log \frac{|S|}{n_c}) = O(\log\log\log |S|)$. As there are at most $O(|S|/\log\log |S|)$ such symbols, the total space spent on them is $O(|S| \log\log\log |S| / \log\log |S|) = o(|S|)$. The mmphfs of the low-frequency symbols, instead, can require $O(\log\log |S|)$ bits per symbol; however, there can only be, in total, $O(|S|/(\log\log |S|)^2)$ occurrences of those symbols. The reason is that each such symbol contributes $\log \frac{|S|}{n_c} > (\log\log |S|)^2$ to $H_0(S) = O(1)$.

[4]Barbay et al. [2010] prove that $\sigma_\ell < 2^{1/\log |S|} |s_\ell| / n_c$, where ℓ is the class to which symbol c is assigned, s_ℓ is the string of the symbols of class ℓ, and σ_ℓ is the number of symbols c assigned to class ℓ. Thus, if $n_c > |S|/\log |S|$, it holds $\sigma_\ell < 2^{1/\log |S|} |s_\ell| \log |S| / |S| \leq \log |S|$. On the other hand, the symbols of classes where $\sigma_\ell \leq \log |S|$ are represented in a constant-time wavelet tree.

Therefore, when $H_0(S) = O(1)$, we can make the total space of the mmphfs $o(|S|)$. This yields the main result of this section.

THEOREM 1. *Given a text $T[1,n]$ over alphabet $[1,\sigma]$, one can build an FM-index occupying $nH_k(T) + o(nH_k(T)) + o(n) + O((n \log n)/s + \sigma^{k+1} \log n)$ bits of space for any $k \geq 0$ and $s > 0$, such that counting is supported in time $t_{\mathsf{count}} = O(m \log\log \sigma)$, locating is supported in time $t_{\mathsf{locate}} = O(s)$, and extraction of a substring of T of length ℓ is supported in time $t_{\mathsf{extract}} = O(s + \ell)$.*

For example, choosing $s = \log n \log\log n$ and $k \leq \alpha \log_\sigma(n) - 1$ we get $nH_k(T)) + o(nH_k(T)) + o(n)$ bits of space. In order to improve counting time to $O(m)$, however, we will need a much more sophisticated approach that cannot be combined with this first simple result.

4. FAST LOCATING AND EXTRACTING USING SELECT

Our strategies for achieving $O(m)$ counting time make use of constant-time *select* operation on the sequences, and therefore will be incompatible with Theorem 1. In this section we develop a new technique that achieves linear locating and extracting time using constant-time *select* operations.

Consider the σ^k partitions $S_1, S_2, \ldots$ of T^{bwt} into contexts of length k. This time we represent each partition using variant (1) of Lemma 1, so the total space is $nH_k(T) + o(nH_k(T)) + o(n) + O(\sigma^{k+1} \log n)$ bits. Unlike the case of operation *access*, the use of bitmap E to mark the beginnings of the partitions and the support for local *select* in the partitions is not sufficient to support global *select* on T^{bwt} (recall Section 2.5).

To obtain global *select* we follow the idea of Golynski et al. [2006]. We set up σ bitmaps B_c, $c \in [1,\sigma]$, of total length $n + o(n)$, as $B_c = 10^{n_c(S_1)}10^{n_c(S_2)}\ldots$, where $n_c(S_i)$ is the number of occurrences of c in S_i. As there are overall n 0s and σ^{k+1} 1s across all the bitmaps B_c, all of them can be represented in compressed form [Raman et al. 2002] using $O(\sigma^{k+1} \log n) + o(n)$ bits, answering *rank* and *select* queries in constant time. Now $q = rank_1(select_0(B_c, j)) = select_0(B_c, j) - j$ tells us the partition number where the jth occurrence of c lies in T^{bwt}, and it is the rth occurrence within S_q, where $r = select_0(B_c, j) - select_1(B_c, q)$. Thus we can implement in constant time operation $select_c(T^{bwt}, j) = select_1(E, q) - 1 + select_c(S_q, r)$, since the local *select* operation in S_q takes constant time.

It is known [Lee and Park 2007] that the Ψ function can be simulated on top of T^{bwt} as $\Psi(i) = select_c(T^{bwt}, j)$, where $c = T[A[i]]$ and i is the jth suffix in A starting with c. Therefore we can store bitmap D and string Q of Section 2.3 so as to compute in constant time $r = rank_1(D, i)$, $c = Q[r]$, and $j = i - select_1(D, r) + 1$.

With this representation we have a constant-time simulation of Ψ using an FM-index, and hence we can locate in time $t_{\mathsf{locate}} = O(s)$ and extract a substring of length ℓ of T in time $t_{\mathsf{extract}} = O(s + \ell)$ using $O((n \log n)/s)$ extra space, as explained in Section 2.3. That is, we can reprove Theorem 1 with a constant-time *select* representation, which is compatible with the linear-time counting data structures that are presented next.

5. IMPROVING CHILD OPERATION IN SUFFIX TREES

We now give a result that has independent interest. We improve the time of operation *child* in the CST of Sadakane [2007], from $O(t_{\mathsf{SA}} \log \sigma)$ to $O(t_{\mathsf{SA}})$. This cannot be achieved with any of the existing schemes using less than $\Theta(n \log \sigma)$ extra space. We do it with $O(n \log \log \sigma) \subseteq o(n \log \sigma)$ extra space.

Given any node of degree d, whose d children are labeled with (strings starting with) symbols $c_1, c_2, \ldots, c_d$, we store all of the symbols in a mmphf f_v occupying $O(d \log \log \sigma)$ bits. As the sum of the degrees of all of the nodes in the suffix tree is at most $2n - 1$, the total space usage is $O(n \log \log \sigma)$ bits.

To answer query $child(v, c)$ we evaluate $f_v(c) = i$ and verify that the ith child of v, u, descends by symbol c. If it does, then $u = child(v, c)$; otherwise v has no child labeled c.

LEMMA 5. *Given a suffix tree we can build an additional data structure that occupies $O(n \log \log \sigma)$ bits, so as to support operation $child(v, c)$ in the time required by computing the ith child of v, u, for any given i, plus the time to extract the first symbol of edge (v, u).*

As explained in Section 2.6, Sadakane's CST represents the tree topology using balanced parentheses. If we use, say, the parentheses representation of Sadakane and Navarro [2010], then the ith child of node v is computed in constant time, as well as all the other operations on parentheses needed in Sadakane's CST. On the other hand, computing the first symbol of edge (v, u) takes time $O(t_{\mathsf{SA}})$. Therefore, we implement $child(v, c)$ in time $O(t_{\mathsf{SA}})$ at the price of $O(n \log \log \sigma)$ extra bits.

Sadakane's CST space is $|CSA| + O(n)$ bits, where $|CSA|$ is the size of the underlying self-index. This new variant raises the space to $|CSA| + O(n \log \log \sigma)$, which is higher when $\sigma = \omega(1)$. However, in this case, the new extra space is well within the usual $o(n \log \sigma)$ bits of redundancy of most underlying CSAs (though not all [Barbay et al. 2010]).

6. IMPROVING COUNTING TIME IN COMPRESSED SUFFIX TREES

Using the encoding of the *child* operation described in the previous section we can find the suffix array interval $A[sp, ep]$ corresponding to a pattern $P[1, m]$ in time $O(m \cdot t_{\mathsf{SA}})$. We show now how to enhance the suffix tree structure with $O(n \log t_{\mathsf{SA}})$ additional bits of space so that this operation requires just the time for extracting m symbols from T given its pointer from A.

To achieve this complexity we store, in addition to the tree topology and to the data structure of the previous section, the length of the string labeling the edge arriving at each node, whenever this number is smaller than $t_{\mathsf{SA}} - 1$. If it is larger than that, then we store a special marker.

Given a pattern P, we traverse the suffix tree top-down. Each time we arrive at a node u and we are inspecting symbol c in P, we use the result of the previous section to find the child $v = child(u, c)$ descending by a string starting with c, and continue the traversal from v. We do not, however, spend $O(t_{\mathsf{SA}})$ time to verify that the mmphf f_c has given us the correct child (i.e., we do not verify that there is a child descending by c), but we just trust the mmphf. Moreover, the other symbols of the string labeling the edge (u, v) are not verified but just skipped in P.

For skipping the characters, we need to know the length of the edge. We notice that, whenever that length is below t_{SA}, we can get it directly from v. Otherwise, we compute it in $O(t_{\mathsf{SA}})$ time using the CST of Sadakane [2007], as the string depth of v minus that of u, $depth(v) - depth(u)$. Note that, because we are skipping at least t_{SA} characters, the total time to traverse the trie is $O(m)$ (this is true even if $m < t_{\mathsf{SA}}$ since we know in constant time whether the next skip surpasses the remaining pattern suffix).

Finally, after we have completed the traversal, we need to check whether the obtained result was right or not. For that we need to extract the first m characters of any of the suffixes below the node arrived at, and compare it with P. If they match, we return the computed range, otherwise P does not occur in T.

Lemma 6. *Given a text $T[1,n]$ we can add a data structure occupying $O(n \log t_{\mathsf{SA}})$ bits on top of its CST, so that the suffix array range corresponding to a pattern $P[1,m]$ can be determined within the time to extract a substring of length m from T whose position in the suffix array is known.*

This gives us a first alphabet-independent FM-index. We can use the FM-index of Section 4 and choose any $s = O(\text{polylog}(n))$, so that $\log t_{\mathsf{SA}} = O(\log\log n)$. Such a limitation on s is not a problem because one prefers to have s as small as possible as long as the redundancy is small, and $s = \Theta(\log n)$ suffices to have $O(n)$-bit redundancy.

Theorem 2. *Given a text $T[1,n]$ over alphabet $[1,\sigma]$, one can build an FM-index occupying $nH_k(T) + o(nH_k(T)) + O(n\log\log n + (n\log n)/s + \sigma^{k+1}\log n)$ bits of space for any $k \geq 0$ and $s = O(\text{polylog}(n))$, such that counting is supported in time $t_{\mathsf{count}} = O(m)$, locating is supported in time $t_{\mathsf{locate}} = O(s)$, and extraction of a substring of T of length ℓ is supported in time $t_{\mathsf{extract}} = O(s+\ell)$.*

Therefore, we have improved the previous result [Barbay et al. 2010] to reach alphabet-independence. On the negative side, we have increased the redundancy from $o(n)$ to $O(n\log\log n)$. Even if $n\log\log n \in o(n\log\sigma)$ whenever $\log\sigma = \omega(\log\log n)$ (and otherwise the problem is already solved [Ferragina et al. 2007]), this increase is unsatisfactory. In the next section we present a more sophisticated approach that almost recovers the original redundancy.

7. BACKWARD SEARCH IN LINEAR TIME

We can achieve $O(m)$ time and compressed redundancy by using the suffix tree to do backward search instead of descending in the tree. As explained in Section 2.5, backward search requires carrying out $O(m)$ *rank* operations. We will manage to simulate the backward search with operations *select* instead of *rank*. We will make use of mmphfs to aid in this simulation.

7.1 Weiner Links

The backward step on the suffix array range for $X = P[i+1,m]$ leads to the suffix array range for $cX = P[i,m]$. When cX is represented by an explicit suffix tree node u (and hence that of X, v, is explicit too), this operation corresponds to taking a Weiner link on character $c = P[i]$, $u = wlink(v,c)$ and $v = slink(u)$ (recall Section 2.1). If cX is not represented by an explicit node, then we will call

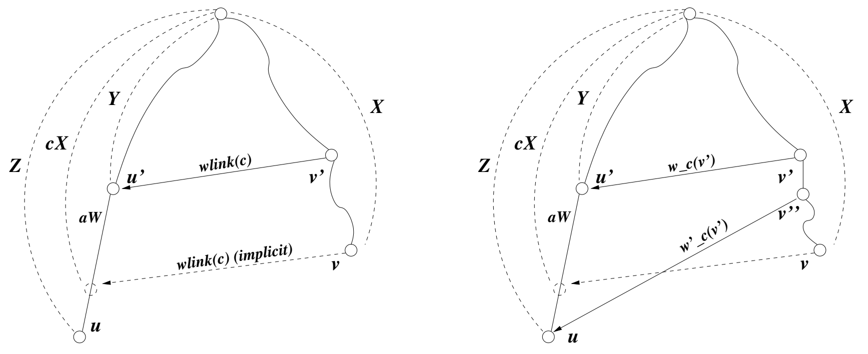

Fig. 1. On the left, relations between suffix tree nodes and the strings they represent, when $wlink(v,c)$ does not exist. Straight lines are suffix tree edges and curved lines are suffix tree paths; arrows represent Weiner links. The dashed node is implicit, and the dashed arrow represents the nonexistent Weiner link. Dashed arcs indicate the strings represented by each node. On the right, the mmphfs w_c and w'_c.

u' the explicit suffix tree node representing the longest prefix of cX, Y, so that $cX = YaW$ for $a \in [1,\sigma]$. Then we call $u = child(u',a)$ in the suffix tree, and u represents a string Z of which cX is a prefix. In such a case, still assuming v is explicit, we call v' its closest ancestor such that $wlink(v',c)$ is defined. It can be seen that, if v' exists, then it holds $u' = wlink(v',c)$. Figure 1 (left) illustrates all these relations.

LEMMA 7. *Let v, u', u be as discussed. Let v' be the closest ancestor of v with $wlink(v',c)$ defined. Then, if v' exists, it holds $v' = slink(u')$ and $u' = wlink(v',c)$.*

PROOF. Let v represent X and u' represent Y. Let v' represent the prefix X' of X, then $u'' = wlink(v',c)$ is an explicit node that represents cX'. If $|cX| > |cX'| > |Y|$, then u'' is between u and u', contradicting the definition of u'. If $|cX'| < |Y|$ then $slink(u')$ is between v and v', contradicting the definition of v'. Then it must be $|cX'| = |Y|$, and thus $u'' = u'$. $\square$

We use the CST of T [Sadakane 2007], so that each node is identified by its preorder value in the parentheses sequence (recall Section 2.6). We use mmphfs to represent the Weiner links. For each symbol $c \in [1,\sigma]$ we create a mmphf w_c and traverse the subtree T_c rooted at $child(root,c)$. As we traverse the (explicit) nodes of T_c in preorder, the suffix links lead us to suffix tree nodes also in preorder (as the strings remain lexicographically sorted after removing their first c). By storing all those suffix link preorders in function w_c, we have that $w_c(v)$ gives in constant time $wlink(v,c)$ if it exists, and an arbitrary value otherwise. More precisely, w_c gives preorder numbers within T_c; it is very easy to convert them to global preorder numbers.

Assume now we are in a suffix tree node v corresponding to suffix array interval $A[sp,ep]$ and to pattern suffix $X = P[i+1,m]$. We wish to determine if the Weiner link $wlink(v,c)$ exists for $c = P[i]$. We can compute $w_c(v) = u$, so that *if* the Weiner link exists, then it leads to node u.

We can determine whether u is the correct Weiner link as follows. First, and assuming the preorder of u is within the bounds corresponding to T_c, we use the

CST to obtain in constant time the range $A[sp', ep']$ corresponding to u [Sadakane 2007]. Now we want to determine if the backward step with c from $A[sp, ep]$ leads us to $A[sp', ep']$ or not. Lemma 8 shows how this can be done using four *select* operations.

LEMMA 8. *Let $A[sp, ep]$ be the suffix array interval for string X, then $A[sp', ep']$ is the suffix array interval for string cX iff*

$$\begin{aligned} select_c(T^{bwt}, i-1) &< sp \quad \wedge \quad select_c(T^{bwt}, i) \geq sp, \text{ and} \\ select_c(T^{bwt}, j) &\leq ep \quad \wedge \quad select_c(T^{bwt}, j+1) > ep, \end{aligned}$$

where $i = sp' - C[c]$, $j = ep' - C[c]$, $C[c]$ is the number of occurrences of symbols $< c$ in the text T, and T^{bwt} is the BWT of T.

PROOF. Note the range of A corresponding to the suffixes that start with symbol c begins at $A[C[c] + 1]$. Then $A[sp']$ is the ith suffix starting with c, and $A[ep']$ is the jth. The classical backward search formula (Section 2.5) for sp' is given next; then we transform it using *rank/select* inequalities.

$$\begin{aligned} & sp' = C[c] + rank_c(T^{bwt}, sp-1) + 1 \\ \Leftrightarrow \quad & i - 1 = rank_c(T^{bwt}, sp-1) \\ \Leftrightarrow \quad & select_c(T^{bwt}, i-1) \leq sp - 1 \quad \wedge \quad select_c(T^{bwt}, i) \geq sp. \end{aligned}$$

The formula for ep' is similar.

$$\begin{aligned} & ep' = C[c] + rank_c(T^{bwt}, ep) \\ \Leftrightarrow \quad & j = rank_c(T^{bwt}, ep) \\ \Leftrightarrow \quad & select_c(T^{bwt}, j) \leq ep \quad \wedge \quad select_c(T^{bwt}, j+1) > ep. \end{aligned}$$

□

Thus we have shown how, given a CST node v, we can compute $wlink(v, c)$ or determine it does not exist, in time $O(t_{\mathsf{select}})$. Now we describe a backward search process on the suffix tree instead of on the suffix array ranges.

7.2 The Traversal

We start at the tree root with the empty suffix $P[m + 1, m]$. In general, being at tree node v corresponding to suffix $X = P[i + 1, m]$, we look for $u = wlink(v, c)$ for symbol $c = P[i]$. If it exists, then we have found node u corresponding to pattern suffix $cX = P[i, m]$ and we are done for that iteration.

If there is no Weiner link from v, it might be that cX is not a substring of T and the search should terminate. However, as explained, it might also be that there is no explicit suffix tree node for cX, but it falls between node u' representing a prefix Y of cX ($cX = YaW$) and node $u = child(u', a)$ representing string Z, of which cX is a prefix.

Our goal is to find node u, which corresponds to the same suffix array interval of cX. For this sake we consider the parent of v, its parent, and so on, until finding the nearest ancestor v' such that $u' = wlink(v', c)$ exists.[5] If we reach, and consider,

[5]Actually we could by chance get the right range $A[sp', ep']$ from an incorrect node, but this would just speed up the algorithm by finding u ahead of time.

the root without finding a Weiner link, then cX is not in T, and neither is P. Else, once we have found u', we compute $u = child(u', a)$ and finish. The correctness of this procedure stems from Lemma 7.

However, computing *child* would be too slow for our purposes, as we require to complete a step in constant time. Instead, we precompute it using a new mmphf w'_c, as follows. For each node $u = child(u', a)$ in T_c, we compute $v' = slink(u')$ and store $v'' = child(v', a)$ in w'_c; note each such v'' is stored exactly once in w'_c. Moreover, the preorders of v'' follow the same order of u. Thus, whenever $u' = wlink(v', c)$, we have the desired child in $w'_c(child(v', a)) = u$. Figure 1 (right) illustrates this.

Now, if $wlink(v, c)$ does not exist, we traverse v and its successive ancestors v' looking for $w'_c(v')$. This will eventually reach node u, so we verify correctness of the mmphf values by comparing directly with the suffix array interval of v, using Lemma 8. Note this test also establishes that cX is a prefix of Z. Only the suffix tree root cannot be dealt with w'_c, but we can easily precompute the σ nodes $child(root, c)$.

Actually only function w'_c suffices. Assume $wlink(v, c) = u$ exists. Then consider u', the parent of u. There will also be a Weiner link from an ancestor v' of v to u'. This ancestor will have a child v'' that points to $w'_c(v'') = u$, and either $v'' = v$ or v'' is an ancestor of v. So we do not check for $wlink(v, c)$ but directly v and its ancestors using w'_c. The mmphf w_c does not need to be built nor used.

7.3 Time and Space

The total number of steps amortizes to $O(m)$: Each time we go to the parent, the depth of our node in the suffix tree decreases. Each time we move by a Weiner link, the depth increases at most by 1, since for any branching node in the path to $u' = wlink(v', c)$ there is a branching node in the path to v'. Since we compute m Weiner links, the total number of operations is $O(m)$. All the operations in the CST tree topology take constant time, and therefore the time t_{select} dominates. Hence the overall time is $O(m \cdot t_{\mathsf{select}})$.

As for the space, the subtree T_c contains n_c leaves and at most $2n_c$ nodes (where n_c is the number of occurrences of symbol c in the text); thus mmphf w'_c stores at most $2n_c$ values in the range $[1, 2n]$. Therefore it requires space $O(n_c(\log\log \frac{n}{n_c} + 1))$ bits, which added over all $c \in [1, \sigma]$ gives a total of $O(n(\log H_0(T) + 1))$, by Lemma 2.

In order to further reduce this space, we partition the mmphfs according to the $O(\sigma^k)$ partitions of the BWT. As in Section 2.4, consider all the possible context strings W_i of length k,[6] their suffix tree nodes v_i, and their corresponding suffix array intervals $A[sp_i, ep_i]$. The corresponding BWT partition is thus $S_{W_i} = S_i = T^{bwt}[sp_i, ep_i]$, of length $n_i = n(S_i) = |S_i| = ep_i - sp_i + 1$. We split each function w'_c into $O(\sigma^k)$ subfunctions w^i_c, each of which will only store the suffix tree preorders that correspond to nodes descending from v_i. There are at most $2n_i$ consecutive preorder values below node v_i, thus the universe of the mmphf w^i_c is of size $O(n_i)$. Moreover, the links stored at w^i_c depart from the subtree that descends from string $c\,W_i$, whose number of leaves is the number of occurrences of c in S_i, $n_c(S_i)$. Thus the total space of all the mmphfs is

[6]Recall from Section 2.5 that the compression booster admits a more flexible partition into suffix tree nodes; but we can choose this way for simplicity of exposition.

$\sum_{c,i} O(n_c(S_i)(\log\log \frac{n_i}{n_c(S_i)} + 1)) = O(n(\log H_k(T) + 1))$, by using Lemma 2 on each S_i and then Lemma 3.

Note there are $O(\sigma^k)$ nodes with context shorter than k. A simple solution is to make a "partition" for each such node, increasing the space by $O(\sigma^k \log n)$. It is easy, along our backward search, to know the context W_i we are in, and thus know which mmphf to query.

By combining the results of Section 4, using a sequence representation with $t_{\mathsf{select}} = O(1)$, with our backward counting algorithm, we have the final result.

THEOREM 3. *Given a text $T[1,n]$ over alphabet $[1,\sigma]$, one can build an FM-index occupying $nH_k(T) + o(nH_k(T)) + O(n + (n\log n)/s + \sigma^{k+1}\log n)$ bits of space for any $k \geq 0$ and $s > 0$, such that counting is supported in time $t_{\mathsf{count}} = O(m)$, locating is supported in time $t_{\mathsf{locate}} = O(s)$, and extraction of a substring of T of length ℓ is supported in time $t_{\mathsf{extract}} = O(s + \ell)$.*

For example, by choosing $s = \log n$ and $k \leq \log_\sigma n - \log_\sigma \log n$, we get $nH_k(T) + o(nH_k(T)) + O(n)$ bits of space.

8. BACKWARD STEP IN $O(1)$ TIME

In the previous section we have shown that the m backward steps used in a backward search for counting occurrences of a pattern $P[1,m]$ can be done in $O(m \cdot t_{\mathsf{select}})$ time. However, any individual backward step can take as much as $O(m \cdot t_{\mathsf{select}})$ time. In this section we show that a data structure occupying $O(n \log\log\sigma)$ bits of space is sufficient to support any backward step in time $O(t_{\mathsf{select}})$.

To get this result, we augment the data structure described in Section 7 with *implicit* Weiner links, that is, those leaving from an explicit suffix tree node but leading to an implicit one (this can be visualized as pointing within an edge, see Figure 1 left).

To get our result, we use the same encoding as in Section 7. That is, for a subtree T_c rooted at $\mathit{child}(\mathit{root}, c)$ we create a mmphf w_c that stores all the preorder numbers of the nodes that are sources of Weiner links with destination in T_c. However, this time we also store the sources of implicit Weiner links.

Now we note that, for a given node u in T_c, we always have an explicit Weiner link pointing to it, but potentially one or more implicit Weiner links pointing within the edge that leads to u. We store in w_c all those sources as pointing to u. As a consequence, the mmphf w_c will store repeated entries that must be dealt with.

We store a bitmap B_c that, for each node of T_c listed in preorder, stores a 1 followed by as many zeros as the number of Weiner links (including implicit ones) pointing to that node (or to the edge leading to the node). Then the mmphf w_c can be thought of as pointing to the zeros of B_c. In order to recover the destination of the Weiner link $\mathit{wlink}(v, c)$ starting from a node v and labeled with character c, we first compute $w_c(v)$, which gives us a virtual location x. Then the final answer will be the node with preorder number $\mathit{select}_0(B_c, x) - x$. This will give us the destination of the Weiner link node if it exists, or garbage otherwise. We still need to check the final result using Lemma 8. Hence we do not need to carry out the upward traversal looking for an ancestor v' of v where $w'_c(v')$ is defined, but we simply use $w_c(v)$.

The number of Weiner links, including implicit ones, is known to be $O(n)$. We include a proof for completenes[7].

LEMMA 9. *There are $O(n)$ implicit Weiner links.*

PROOF. Leaves can have only one (implicit or explicit) Weiner link. Consider then an internal suffix tree node v, representing string X, and an implicit Weiner link leaving from v. Since v is internal, the text must contain at least two strings, Xa and Xb. Now consider the destination u of the implicit Weiner link $wlink(v,c) = u$, so u represents cX. Since u is an implicit node, there is a string cXd in the text for only one possible d, so that either $d \neq a$ or $d \neq b$. This means that one of the occurrences of Xa and Xb in the text is not preceded by c, but rather by some $e \neq c$. That means that the text contains two disticnt strings eX and cX. Now consider the suffix tree built on the reverse text. It must contain an internal node v^r representing string X with at least two children, labeled with strings starting with c and e. Therefore, there is an edge from v^r to $child(v^r, c)$. This edge is an injective function of v and c. Thus, since the suffix tree of the reverse text has $O(n)$ edges, there are $O(n)$ implicit Weiner links $wlink(v,c)$ leaving from internal nodes v of the suffix tree of the text. □

As a consequence, the space for all the B_c bitmaps is $O(n)$ bits. As for the functions w_c, we have that the total number of Weiner links is $O(n)$. These are partitioned into σ functions w_c, each of them containing an arbitrary number of elements n'_c (not necessarily $n_c = |T_c|$ because of the repeated nodes). Thus the total space used by all the w_c mmphfs is bounded by $\sum_c O(n'_c(\log\log \frac{n}{n'_c} + 1))$ bits, which by convexity of the logarithm is $O(n \log\log\sigma)$ bits, as $\sum n'_c = n$.

Using again a constant-time *select* self-index as in the previous section, we obtain the following extended result.

THEOREM 4. *Given a text $T[1,n]$ over alphabet $[1,\sigma]$, one can build an FM-index occupying $nH_k(T)+o(nH_k(T))+O(n\log\log\sigma+(n\log n)/s+\sigma^{k+1}\log n)$ bits of space for any $k \geq 0$ and $s > 0$, such that counting is supported in time $t_{\mathsf{count}} = O(m)$, locating is supported in time $t_{\mathsf{locate}} = O(s)$, and extraction of a substring of T of length ℓ is supported in time $t_{\mathsf{extract}} = O(s+\ell)$. A backward step that starts from a valid interval can be computed in $O(1)$ time.*

In particular, this gives constant-time support to the Weiner link operation on CSTs. For example that of Sadakane [2007] takes $O(\log n)$ time on top of its original CSA [Sadakane 2003]. Our new compressed suffix array representation gives constant time for computing Ψ and LF, and thus constant-time support for both suffix and Weiner links.

9. CONSTANT-ACCESS REPRESENTATIONS REVISITED

We have achieved our best results with sequence representations that support constant-time *select*, and left the result of Section 3 as a warmup exercise. In this section we consider the situation where the sequence representation must give constant-time *access* (and *select* in time $O(\log\log\sigma)$), due to other restrictions.

[7]We thank Roman Kolpakov for explaining it to us.

We show how we can achieve $O(m+s)$ counting time, at the price of $O(n \log\log \sigma)$ bits of redundancy.

We use the technique of Section 8, yet without checking the correctness of each backward step. Since we have added all the necessary Weiner links, either the mmphf $w'_c(v)$ gives us the correct answer or there is no occurrence of cX in the text, where v represents string X. Thus we can carry out all the steps and only at the end we need to check that the range we have obtained is correct or not.

The range is correct if and only if pattern P occurs in T, so all we have to do is to extract any occurrence in the range and compare it with P. This is done in time $O(m+s)$ using *locate* and *extract*.

THEOREM 5. *Given a text $T[1,n]$ over alphabet $[1,\sigma]$, one can build an FM-index occupying $nH_k(T) + o(nH_k) + O(n \log\log \sigma + (n \log n)/s + \sigma^{k+1} \log n)$ bits of space for any $k \geq 0$ and $s > 0$, such that counting is supported in time $t_{\mathsf{count}} = O(m+s)$, locating is supported in time $t_{\mathsf{locate}} = O(s)$, and extraction of a substring of T of length ℓ is supported in time $t_{\mathsf{extract}} = O(s+\ell)$.*

10. CONCLUSIONS

This article contributes to bridging the gap in the time performances of classical and compressed text indexes. While the former use $O(n \log n)$ bits to index a text T of length n, they offer time performances that are independent of the alphabet size σ. Compressed indexes, instead, use space asymptotically equal to that of the compressed text, but their time performances worsen as the alphabet size grows.

Our results show that it is possible to retain compressed space while achieving query times that are independent of the alphabet size, and that depend only on the input/output size plus a sampling step s that allows one trade time for space. Moreover, in our main result (Theorem 3), we have also removed the dependence on the alphabet size from the redundancy space of the compressed indexes. That is, the redundancy is sublinear in the compressed text size ($o(nH_k(T))$) and not in the plain text size ($o(n \log \sigma)$). The only aspect of the result that depends on σ is the extra redundancy term $O(\sigma^{k+1} \log n)$. However, this rather limits k and it is essentially unavoidable under the kth order empirical entropy model [Gagie 2006]. Therefore, we have achieved *full alphabet independence* within compressed space.

Our results carry over compressed suffix trees (CSTs), which have much richer functionality than text indexes [Sadakane 2007; Fischer et al. 2009; Fischer 2010; Russo et al. 2011]. CSTs are generally alphabet-independent, but depend on an underlying self-index. Thus our new alphabet-independent self-indexes immediately yield alphabet-independent CSTs. The only exception was the *child* query of Sadakane [2007], which we have also made alphabet-independent in this article.

This does not mean that there are no further challenges in this area; actually there are many. Some open problems are:

(1) Compared with the previously least alphabet-sensitive compressed self-index [Barbay et al. 2010], we have achieved full alphabet independence at the price of converting an $o(n)$-bits term in their space redundancy into $O(n)$. It is open whether one can achieve full alphabet independence within $o(n)$ redundancy. The difference is important for not-so-large alphabets.

(2) Our times are linear in the input (i.e., $O(m)$ for counting the occurrences of a pattern of length m) or in the output (i.e., $O(\ell)$ time for displaying ℓ consecutive text symbols), plus $O(s)$. However, some classical indexes achieve *optimal* times in the RAM model, $O(m/\log_\sigma n)$ and $O(\ell/\log_\sigma n)$, respectively. It is open whether one can achieve optimal times within optimal space (say, under the $H_k(T)$ entropy model). Grossi et al. [2003] have partially achieved this goal, using $cnH_k(T)$ bits for any $c > 2$, and with a polylogarithmic additive penalty in both times.

(3) Spending $O((n \log n)/s)$ bits of redundancy and having additive penalties of $O(s)$ for all the operations is not necessarily the best possible use of the space. There exists an incomparable alternative [Grossi et al. 2003] that uses $cnH_k(T)$ space and supports, for example, locating in time $O(\log_\sigma^{1/(c-1)} n + \log \sigma)$. Is there a storage model obtaining the best from both, or doing even better?

(4) Finally, some of the results we obtained, such as faster suffix tree traversals, constant-time backward steps, or linear counting time on constant-access sequence representations, require $o(n \log \sigma)$ bits of redundancy. It is open whether this can be reduced to sublinear on the compressed text size, that is, to $o(nH_k(T))$ bits.

REFERENCES

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series (1985), pp. 85–96. Springer-Verlag.

Baeza-Yates, R. and Gonnet, G. H. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM 43*, 6, 915–936.

Baeza-Yates, R. and Ribeiro-Neto, B. 2011. *Modern Information Retrieval* (2nd ed.). Addison-Wesley.

Barbay, J., Gagie, T., Navarro, G., and Nekrich, Y. 2010. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st ISAAC* (2010), pp. 315–326. Part II.

Barbay, J., He, M., Munro, J. I., and Rao, S. S. 2007. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th SODA* (2007), pp. 680–689.

Belazzougui, D., Boldi, P., Pagh, R., and Vigna, S. 2009a. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *Proc. 20th SODA* (2009), pp. 785–794.

Belazzougui, D., Boldi, P., Pagh, R., and Vigna, S. 2009b. Theory and practise of monotone minimal perfect hashing. In *Proc. 10th ALENEX* (2009).

Belazzougui, D. and Navarro, G. 2012. New lower and upper bounds for representing sequences. In *Proc. 20th ESA*, LNCS 7501 (2012), pp. 181–192.

Beller, T., Gog, S., Ohlebusch, E., and Schnattinger, T. 2011. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *Proc. 18th SPIRE* (2011), pp. 197–208.

Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

Clark, D. 1996. *Compact Pat Trees.* Ph. D. thesis, University of Waterloo, Canada.

Crochemore, M. and Rytter, W. 2003. *Jewels of Stringology.* World Scientific.

Farach, M. 1997. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS* (1997), pp. 137–143.

Ferragina, P., Giancarlo, R., Manzini, G., and Sciortino, M. 2005. Boosting textual compression in optimal linear time. *Journal of the ACM 52*, 4, 688–713.

Ferragina, P., González, R., Navarro, G., and Venturini, R. 2009. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics 13*, article 12.

Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In *Proc. 41st FOCS* (2000), pp. 390–398.

Ferragina, P. and Manzini, G. 2005. Indexing compressed text. *Journal of the ACM 52*, 4, 552–581.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3*, 2, article 20.

Fischer, J. 2010. Wee LCP. *Information Processing Letters 110*, 317–320.

Fischer, J., Mäkinen, V., and Navarro, G. 2009. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science 410*, 51, 5354–5364.

Gagie, T. 2006. Large alphabets and incompressibility. *Information Processing Letters 99*, 6, 246–251.

Golynski, A. 2009. Cell probe lower bounds for succinct data structures. In *Proc. 20th SODA* (2009), pp. 625–634.

Golynski, A., Munro, J. I., and Rao, S. S. 2006. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA* (2006), pp. 368–373.

Gonnet, G., Baeza-Yates, R., and Snider, T. 1992. *Information Retrieval: Data Structures and Algorithms*, Chapter 3: New indices for text: Pat trees and Pat arrays, pp. 66–82. Prentice-Hall.

Grossi, R., Gupta, A., and Vitter, J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th SODA* (2003), pp. 841–850.

Grossi, R., Orlandi, A., and Raman, R. 2010. Optimal trade-offs for succinct string indexes. In *Proc. 37th ICALP* (2010), pp. 678–689.

Grossi, R. and Vitter, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd STOC* (2000), pp. 397–406.

Gusfield, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th FOCS* (1989), pp. 549–554.

Kärkkäinen, J. and Sanders, P. 2003. Simple linear work suffix array construction. In *Proc. 30th ICALP*, LNCS 2719 (2003), pp. 943–955.

Lam, T.-W., Sung, W.-K., Tam, S.-L., Wong, C.-K., and Yiu, S.-M. 2008. Compressed indexing and local alignment of DNA. *Bioinformatics 24*, 6, 791–797.

Lee, S. and Park, K. 2007. Dynamic rank-select structures with applications to run-length encoded texts. In *Proc. 19th CPM* (2007), pp. 95–106.

Mäkinen, V., Välimäki, N., Laaksonen, A., and Katainen, R. 2010. Unified view of backward backtracking in short read mapping. In T. Elomaa, H. Mannila, and P. Orponen Eds., *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060 (2010), pp. 182–195. Springer.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22*, 5, 935–948.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48*, 3, 407–430.

McCreight, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2, 262–272.

Munro, I. 1996. Tables. In *Proc. 16th FSTTCS* (1996), pp. 37–42.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39*, 1, article 2.

Ohlebusch, E., Gog, S., and Kügel, A. 2010. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. 17th SPIRE* (2010), pp. 347–358.

Raman, R., Raman, V., and Rao, S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA* (2002), pp. 233–242.

Russo, L., Navarro, G., and Oliveira, A. 2011. Fully-compressed suffix trees. *ACM Transactions on Algorithms 7*, 4, article 53.

Sadakane, K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th ISAAC* (2000), pp. 410–421.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms 48*, 2, 294–313.

Sadakane, K. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems 41*, 4, 589–607.

Sadakane, K. and Navarro, G. 2010. Fully-functional succinct trees. In *Proc. 21st SODA* (2010), pp. 134–149.

Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica 14*, 3, 249–260.

Weiner, P. 1973. Linear pattern matching algorithm. In *Proc. 14th Ann. IEEE Symp. on Switching and Automata Theory* (1973), pp. 1–11.