Alphabet-Independent Compressed Text Indexing*

Djamal Belazzougui[1] and Gonzalo Navarro[2]

[1] LIAFA, Univ. Paris Diderot - Paris 7, France
dbelaz@liafa.jussieu.fr
[2] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract. Self-indexes can represent a text in asymptotically optimal space under the k-th order entropy model, give access to text substrings, and support indexed pattern searches. Their time complexities are not optimal, however: they always depend 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 on compressed suffix trees.

1 Introduction

Text indexes, like the suffix tree [1] and the suffix array [18], can *count* the occurrences of a pattern $P[1,m]$ in a text $T[1,n]$ over alphabet $[1,\sigma]$ in time $t_{\mathsf{count}} = O(m)$ or even $t_{\mathsf{count}} = O(m/\lg_\sigma n)$ (suffix trees), or $t_{\mathsf{count}} = O(m + \lg n)$ (suffix arrays). Afterwards, 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 substring $T[i, i+\ell-1]$ in optimal time $t_{\mathsf{extract}} = O(\ell/\lg_\sigma n)$. Yet, their $O(n \lg n)$-bit space complexity renders these structures unapplicable for large text collections.

Compressed text *self-indexes* [21] represent a text $T[1,n]$ over alphabet $[1,\sigma]$ within compressed space and allow not only extracting any substring of T, but also counting and locating the occurrences of patterns.

A popular model to measure text compressibility is the *k-th order empirical entropy* [19], $H_k(T)$. This 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 (or follow) the symbol to encode. It holds $0 \leq H_k(T) \leq H_{k-1}(T) \leq \lg \sigma$.

Starting with the FM-index [9] and the Compressed Suffix Array [16,23], self-indexes have evolved up to a point where they have reached asymptotically optimal space within the k-th order entropy model, that is, $nH_k(T) + o(n \lg \sigma)$ bits [24,14,10,21,11,3,2]. While remarkable in terms of space, self-indexes have not retained the time complexities of the classical suffix trees and arrays.

* Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile. First author also partially supported by the French ANR-2010-COSI-004 MAPPI Project.

C. Demetrescu and M.M. Halldórsson (Eds.): ESA 2011, LNCS 6942, pp. 748–759, 2011.

Table 1 lists the current space-optimal self-indexes. All follow a model where a sampling step s is chosen (which costs $O((n \lg n)/s)$ bits, so at least we have $s = \omega(\lg_\sigma n)$ 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. There are some recent results [2] where the concept of asymptotic optimality is carried out one step further, achieving $o(nH_k(T)) + o(n) \subseteq o(n \lg \sigma)$ extra space. The only structure achieving locating and extracting times independent of σ is Sadakane's [24], yet its counting time is the worst. Note that a recent FM-index [11] achieves $O(m)$ counting, $O(s)$ locating, and $O(s + \ell)$ extraction time when the alphabet is polylogarithmic in the text size, $\sigma = O(\text{polylog}(n))$.

Only the structures of Grossi et al. [14] escape from this general scheme, however they need to use more than the optimal space in order to achieve alphabet independent times. By using $(2 + \varepsilon)nH_k(T) + o(n \lg \sigma)$ bits, for any $\varepsilon > 0$, they achieve the optimal $O(m/\lg_\sigma n)$ counting time, albeit with an additive polylogarithmic penalty of $p(n) = O(\lg_\sigma^{(3+\varepsilon)/(1+\varepsilon)} n \lg^2 \sigma)$. They can also achieve sublogarithmic locating time, $O(\lg^{1/(1+\varepsilon)} n)$. Finally the extraction time is also optimal plus the polylogarithmic penalty, $O(\ell/\lg_\sigma n + p(n))$.

Table 1. Current and our new complexities for self-indexes, for the case $\lg \sigma = \omega(\lg \lg n)$. The space results (in bits) hold for any $k \leq \alpha \lg_\sigma(n) - 1$ 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 [24] refers to a more recent analysis [21]; see also the clarifications in `www.dcc.uchile.cl/gnavarro/fixes/acmcs06.html`.

Source	Space $(+O((n \lg n)/s))$	Counting	Locating	Extracting
[14]	$nH_k + o(n \lg \sigma)$	$O(m \lg \sigma + \lg^4 n)$	$O(s \lg \sigma)$	$O((s + \ell) \lg \sigma)$
[24]	$nH_k + o(n \lg \sigma)$	$O(m \lg n)$	$O(s)$	$O(s + \ell)$
[11]	$nH_k + o(n \lg \sigma)$	$O(m \frac{\lg \sigma}{\lg \lg n})$	$O(s \frac{\lg \sigma}{\lg \lg n})$	$O((s + \ell) \frac{\lg \sigma}{\lg \lg n})$
[3]	$nH_k + o(n \lg \sigma)$	$O(m \lg \lg \sigma)$	$O(s \lg \lg \sigma)$	$O((s + \ell) \lg \lg \sigma)$
[2]	$nH_k + o(nH_k) + o(n)$	$O(m \frac{\lg \sigma}{\lg \lg n})$	$O(s \frac{\lg \sigma}{\lg \lg n})$	$O((s + \ell) \frac{\lg \sigma}{\lg \lg n})$
[2]	$nH_k + o(nH_k) + o(n)$	$O(m \lg \lg \sigma)$	$O(s \lg \lg \sigma)$	$O((s + \ell) \lg \lg \sigma)$
Ours	$nH_k + o(nH_k) + O(n)$	$O(m)$	$O(s)$	$O(s + \ell)$

Our main result in this paper is the last row of Table 1. We achieve for the first time *full alphabet independence* for all alphabet sizes, at the price of converting an $o(n)$-bit redundancy into $O(n)$. This is an important step towards leveraging the time penalties incurred by asymptotically optimal space indexes.

We apply various techniques to achieve our result. The general strategy is to find an alternative to the use of *rank* operation on sequences, on which all FM-based indexes build, and for which no constant-time solution is known. We combine FM-indexes with concepts of Compressed Suffix Arrays, monotone minimum perfect hash functions, and compressed suffix trees. As a byproduct we enhance Sadakane's compressed suffix tree [25], which uses $O(n)$ bits on top of an underlying self-index, with a data structure using $O(n \lg \lg \sigma)$ bits that

speeds up the important *child* operation; the only one that still depended on the alphabet size and now is also freed from that dependence.

Sections 2 and 3 give the necessary background on self-indexes and monotone minimal perfect hash functions (mmphfs). The latter section finishes with a simple illustration of the power of mmphfs to achieve alphabet independence on locating and extracting time on FM-indexes. This is not in the main path to achieve alphabet independence on counting as well, however, so in Section 4 we reimplement locating and extracting using constant-time *select* operations. Section 5 shows how to use mmphfs to improve the *child* operation on suffix trees, and this is used in Section 6 to reduce the search time on suffix trees. These results are of general interest, but are not used in Section 7, where we use (compressed) suffix trees in a different way to finally achieve linear counting time (in combination with the results of Section 4).

2 Compressed Self-indexes

An important subproblem that arises in 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}.
- $rank_c(S,i)$ is the number of times symbol c appears in $S[1,i]$, in time t_{rank}.
- $select_c(S,i)$ is the position in S of the ith occurrence of c, in time t_{select}.

For the particular case of bitmaps, constant-time operations can be achieved using $n + o(n)$ bits [20], or $\lg \binom{n}{m} + O(\lg\lg m) + o(n) = nH_0(S) + O(m) + o(n)$ bits, where m is the number of 1s (or 0s) in S [22]. General sequences can also be represented within asympotically zero-order entropy space $nH_0(S) = \sum_{c\in[1,\sigma]} n_c \lg \frac{n}{n_c}$, where n_c is the number of times c occurs in S. Among the many compressed sequence representations [13,11,3,2,15], we emphasize two results for this paper. The first corresponds to Thm. 1, variant (i), of Barbay et al.'s recent result [2]. The second is obtained by using the same theorem, yet replacing Golynski et al.'s representation [13] for the sequences of similar frequency, by another recent result of Grossi et al. [15] (the scheme compresses itself to $H_k(S)+o(|S|\lg\sigma)$ bits, but with more restrictions; when combining with Barbay et al. we only need that it takes $|S|\lg\sigma + o(|S|\lg\sigma)$ bits).

Lemma 1 ([2,15]). *A sequence $S[1,n]$ over alphabet $[1,\sigma]$ can be represented within $nH_0(S)+o(n(H_0(S)+1))+O(\sigma\lg n)$ bits of space, so that the operations are supported in times either (1) $t_{\mathsf{access}} = t_{\mathsf{rank}} = O(\lg\lg\sigma)$ and $t_{\mathsf{select}} = O(1)$, or (2) $t_{\mathsf{select}} = t_{\mathsf{rank}} = O(\lg\lg\sigma)$ and $t_{\mathsf{access}} = O(1)$.*

The FM-index [10] is a compressed self-index built on such sequence representations. In its modern form [11], the index computes the Burrows-Wheeler transform [6] of a text $T[1,n]$, $T^{bwt}[1,n]$, then cuts it into $O(\sigma^k)$ partitions, and represents each partition as a sequence supporting *rank* and *access* operations. From their analysis [11] it follows that if each such sequence S is represented within $|S|H_0(S)+o(|S|H_0(S))+o(|S|)+O(\sigma\lg n)$ bits of space, then the overall space of the index is $nH_k(T)+o(nH_k(T))+o(n)+O(\sigma^{k+1}\lg n)$. The latter term

is usually removed by assuming $k \leq \alpha \lg_\sigma(n) - 1$ and constant $0 < \alpha < 1$. This is precisely the space Barbay et al. [2] achieve, and the best space reported so far for compressed text indexes under the k-th order entropy model (see Table 1).

A fundamental operation of the FM-index is the so-called *LF-mapping* $LF(i) = C[c] + rank_c(T^{bwt}, i)$, where $c = T^{bwt}[i]$. Here C is a small array storing in $C[c]$ the number of occurrences in T of symbols $< c$. The LF-mapping is used with various purposes. The BWT T^{bwt} is actually aligned with the suffix array [18] $A[1,n]$ of $T[1,n]$, so that $T^{bwt}[i] = T[A[i]-1]$. The suffix array points to all the suffixes of T in lexicographic order, and thus the occurrences of any pattern $P[1,m]$ in T appear in a range of $A[sp,ep]$. The meaning of the LF-mapping is that, if $A[i] = j$, then $A[LF(i)] = j - 1$, that is, it lets us move virtually backwards in T, while using suffix array positions. The FM-index marks the partitions of the BWT in a sparse bitmap P that is represented within $O(\sigma^k \lg n) + o(n)$ bits and offers constant-time *rank* and *select* [22]. Therefore the time to compute the LF-mapping is $t_{\mathsf{LF}} = O(t_{\mathsf{access}} + t_{\mathsf{rank}})$, where t_{access} and t_{rank} refer to the times in the representation of the partitions.

The time to compute LF impacts all the times of the FM-index. By using a sampling step s, which yields extra space $O((n \lg n)/s)$ bits, 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 $O((s+\ell) \cdot t_{\mathsf{LF}})$. As no known solution offers $t_{\mathsf{rank}} = O(1)$, we will circumvent the dependence on t_{rank} in order to achieve $t_{\mathsf{LF}} = O(1)$.

The remaining operation offered by the FM-index is *counting*, that is, determining the area $A[sp,ep]$ where pattern P occurs, so that its occurrences can be counted as $ep - sp + 1$ and each occurrence position can be located using $A[i]$, for $sp \leq i \leq ep$. 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 $rank_c(T^{bwt}, i)$), and therefore achieving linear time for it will require a more elaborate technique.

The other family of self-indexes are Compressed Suffix Arrays (CSAs) [16,24,14]. Here the main component is function $\Psi(i) = A^{-1}[A[i]+1]$, which is the inverse of function LF. The array Ψ is represented directly within compressed space and giving constant access time to any value. A sparse bitmap $D[1,n]$ is stored, so that we mark positions $i = 1$ and the positions i such that $T[A[i]] \neq T[A[i-1]]$. In addition, the distinct symbols of T are stored in a string $Q[1,\sigma]$, in lexicographic order. By storing D in compressed form [22], D and Q occupy $O(\sigma \lg n) + o(n)$ bits and we have constant time *rank* and *select* on D. Then we have $T[A[i]] = Q[rank_1(D,i)]$. Moreover, $T[A[i]+k] = T[A[\Psi^k(i)]]$, which gives any string $T[A[i], A[i]+\ell-1]$ in time $O(\ell)$.

This enables a simple binary-search-based suffix array searching for $P[1,m]$ in time $O(m \lg n)$. By using the same sampling mechanism mentioned for the FM-index, and considering that this time Ψ virtually moves forwards instead of backwards in T, we achieve $O(s)$ locating time and $O(s+\ell)$ extracting time.

For completeness we describe the sampling for the CSA. For locating, sample T regularly every s positions by setting up 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 compute $A[i]$, compute successively $j = \Psi^k(j)$ for $k = 0, 1, \ldots, s-1$ until $V[j] = 1$; then $A[i] = S_A[rank_1(V,j)] \cdot s + k$. For extracting simply store $S_T[j] = A^{-1}[1 + s \cdot j]$ for $j = 0, 1, \ldots, n/s$, then to extract $T[i, i+\ell-1]$, 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.

3 Monotone Minimal Perfect Hash Functions

A *monotone minimal perfect hash function (mmphf)* [4,5] $f : [1,u] \to [1,n]$, for $n \leq u$, 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 $\lg \binom{u}{n}$ bits, more precisely $O(n \lg \lg \frac{u}{n} + n)$ bits. This allows using it to speed up operations while adding an extra space that is asymptotically negligible.

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

The LF-mapping 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 c. Therefore the time of the LF function becomes $O(1)$ if we have constant access time to the BWT. Consider now partitioning the BWT as in the FM-index [11]. Our extra space is $O(|S|(\lg H_0(S) + 1))$ within each partition S of the BWT. By the log-sum inequality again[2] we get total space $O(n(\lg H_k(T) + 1))$. We obtain the following result.

Lemma 2. *By adding $O(n(\lg H_k(T) + 1))$ bits to an FM-index built on text $T[1,n]$ over alphabet $[1,\sigma]$, one can compute the LF-mapping in time $t_{\mathsf{LF}} = O(t_{\mathsf{access}})$, where t_{access} is the time needed to access any element in T^{bwt}.*

[1] Given n pairs of numbers $a_i, b_i > 0$, it holds $\sum a_i \lg \frac{a_i}{b_i} \geq (\sum a_i) \lg \frac{\sum a_i}{\sum b_i}$. Use $a_i = n_c/n$ and $b_i = -a_i \lg a_i$ to obtain the claim.

[2] This time using $a_i = |S_i|$ and $b_i = |S_i| \lg H_0(S_i)$.

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

The sequence representation for each partition S takes $|S|H_0(S)+o(|S|H_0(S)) + o(|S|) + O(\sigma \lg n)$ bits. Added over all the partitions [11], this gives the main space term $nH_k(T) + o(nH_k(T)) + o(n) + O(\sigma^{k+1} \lg n)$, as explained. On top of this, Lemma 2 requires $O(n(\lg H_k(T) + 1))$ bits. This is $o(nH_k(T)) + O(n)$ if $H_k(T) = \omega(1)$, and $O(n)$ otherwise.

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

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. This is what the rest of the paper is about.

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 Thm. 1. In this section we develop a new technique that achieves linear locating and extracting time using constant-time *select* operations.

Consider the $O(\sigma^k)$ partitions of T^{bwt}. This time we represent each partition using variant (1) of Lemma 1, so the total space is $nH_k(T)+o(nH_k(T))+o(n)+ O(\sigma^{k+1} \lg n)$ bits. Unlike the case of *access*, the use of bitmap P to mark the beginnings of the partitions and the support for local *select* in the partitions is not sufficient to achieve global *select* on T^{bwt}.

Following Golynski et al.'s idea [13] we set up σ bitmaps B_c, $c \in [1, \sigma]$, of total length $n + o(n)$, as $B_c = 01^{n(c,1)}01^{n(c,2)} \ldots 01^{n(c,\lceil n/b \rceil)}$, where $n(c, i)$ is the number of occurrences of symbol c in partition S_i. So there are overall n 1s and $O(\sigma^{k+1})$ 0s across all the B_c bitmaps, and thus all of them can be represented in compressed form [22] using $O(\sigma^{k+1} \lg n)$ bits, answering *rank* and *select* queries in constant time. Now $q = rank_0(select_1(B_c, j)) = select_1(B_c, j) - j$ tells us the block number where the jth occurrence of c lies in T^{bwt}, and it is the rth occurrence within S_q, where $r = select_1(B_c, j) - select_0(B_c, q)$. Thus we can implement in constant time operation $select_c(T^{bwt}, j) = select_1(P, q) - 1 + select_c(S_q, r)$, since the local *select* operation in S_q takes constant time.

It is known [17] 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 j-th suffix in A starting with c. Therefore we can use bitmap D and string Q 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 $O((n \lg n)/s)$ extra space, as explained in Section 2. This representation 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. One of the most important and frequently used operations in compressed suffix trees (CSTs) is also usually the slowest: operation $child(v, c)$ gives the node that descends from node v by symbol c, if it exists. For example, if t_{SA} is the time to compute a cell of the underlying suffix array or of its inverse permutation,[3] then operation *child* costs time $O(t_{\mathsf{SA}} \lg \sigma)$ in Sadakane's CST [25].

We improve the operation as follows. Given any node of degree d whose d children are labeled with characters $c_1, c_2, \ldots, c_d$, we store all of them in a mmphf f_v occupying $O(d \lg \lg \sigma)$ 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 $O(n \lg \lg \sigma)$ bits.

To answer $child(v, c)$ we compute $f_v(c) = i$ and verify that the ith child of v, u, descends by symbol c. If so, then $u = child(v, c)$, else v has no child labeled c.

Lemma 3. *Given a suffix tree we can build an additional data structure that occupies $O(n \lg \lg \sigma)$ 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 letter of edge (v, u).*

Sadakane's CST represents the tree topology using balanced parentheses. If we use Sadakane and Navarro's parentheses representation [26], then the ith child of node v is computed in constant time, as well as all the other operations used in Sadakane's CST. Moreovoer, computing the first letter of the edge (v, u) takes time $O(t_{\mathsf{SA}})$. Therefore, we reduce the time for operation $child(v, c)$ from $O(t_{\mathsf{SA}} \lg \sigma)$ to $O(t_{\mathsf{SA}})$ at the price of $O(n \lg \lg \sigma)$ extra bits. Sadakane's CST space is $|CSA| + O(n)$ bits, where $|CSA|$ is the size of the underlying self-index. While this new variant raises the space to $|CSA| + O(n \lg \lg \sigma)$, it turns out that, for $\sigma = \omega(1)$, the new extra space is within the usual $o(n \lg \sigma)$ bits of redundancy of most underlying CSAs (though not all of them [2]).

We note that Sadakane [25] also shows how to achieve time complexity $O(t_{\mathsf{SA}})$ for *child*, but at the much heavier expense of using $O(n \lg \sigma)$ extra space.

6 Improving Counting Time in Compressed Suffix Trees

Using the encoding of the *child* operation as 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

[3] 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.

$O(n \lg t_{\mathsf{SA}})$ extra bits of space so that this operation requires just $O(m)$ time in addition to that for extracting m symbols from T given its pointer from A.

We use a blind search strategy [8]. We first traverse the trie considering only the characters at branching nodes (moreover we can make mistakes, as seen soon). This returns an interval $A[sp, ep]$ whose correctness is then checked at the end. We store, in addition to the tree topology and to the data structure of Section 5, the number of skipped characters 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. Then, given a pattern P, we traverse the suffix tree top-down and each time we have a branching node and we are at character c in the pattern, we use the result of Section 5 to find the child labeled by c (yet we do not spend time in verifying it) and continue the traversal from that child. For skipping the characters during the top-down traversal, we notice that whenever the skip count of a node is below t_{SA}, we can get it from the node, otherwise we get it in $O(t_{\mathsf{SA}})$ time using Sadakane's CST [25], as the string depth of the node minus that of its parent, $depth(v) - depth(parent(v))$. 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). Finally, after we have finished 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 them with P. If they match, we return the computed range, otherwise P does not occur in T.

Lemma 4. *Given a text $T[1, n]$ we can add a data structure occupying $O(n \lg t_{\mathsf{SA}})$ bits on top of its CST, so that the suffix array range corresponding to a pattern $P[1, m]$ can be determined within $O(m)$ time plus 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 choose any $s = O(\text{polylog}(n))$, so that $\lg t_{\mathsf{SA}} = O(\lg \lg n) \subset o(\lg \sigma)$ whenever $\lg \sigma = \omega(\lg \lg n)$ (recall that the other case is already solved [11]).

Theorem 2. *Given a text $T[1, n]$ over alphabet $[1, \sigma]$, one can build an FM-index occupying $nH_k(T) + o(n \lg \sigma) + O((n \lg n)/s + \sigma^{k+1} \lg n)$ bits of space for any $k \geq 0$ and $0 < s = O(\text{polylog}(n))$, 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 in time $t_{\mathsf{extract}} = O(s + \ell)$.*

An unsatisfactory aspect of this theorem is that we have increased the redundancy from $o(nH_k(T)) + O(n)$ to $o(n \lg \sigma)$. In the next section we present a more sophisticated approach that recovers the original redundancy.

7 Backward Search in $O(m)$ 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, 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.

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 corresponds to an explicit (i.e., branching) suffix tree node (and hence that of X is explicit too), this operation corresponds to taking a *Weiner link* [27] on character $c = P[i]$ from the suffix tree node corresponding to $X = P[i+1, m]$. Weiner links are in some sense the inverses of *suffix links*, which lead from the suffix tree node u representing string cX to the node v representing string X, $slink(u) = v$; the Weiner link by c at node v is u, $wlink(v, c) = u$. If cX is not explicit but descends by string aW from its parent u', then X descends by aW from a node v' such that $wlink(v', c) = u'$, and v' is the closest ancestor of v with $wlink(\cdot, c)$ defined.

We use the CST of T [25], so that each node is identified by its preorder value in the parentheses sequence. 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 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 it to global preorder numbers.

Assume now we are in a suffix tree node v corresponding to suffix array interval $A[sp, ep]$ and 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 the range $A[sp', ep']$ corresponding to u [25]. Now we want to determine if the backward step with $P[i]$ from $A[sp, ep]$ leads us to $A[sp', ep']$ or not. Lemma 5 shows how this can be done using four *select* operations.

Lemma 5. *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 that the range of A for 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) for sp' is given next; then we transform it using *rank*/*select* inequalities. The formula for ep' is similar.

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

□

Thus we have shown how, given a CST node v, compute $wlink(v, c)$ or determine it does not exists in time $O(t_{\mathsf{select}})$.[4] Now we describe a backward search process on the suffix tree instead of on the suffix array ranges.

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. If we reach the root without finding a Weiner link, then c is not in T, and neither is P. Once we have found u' we compute $u = child(u', a)$ and we finish.

However, computing *child* would be too slow for our purposes. Instead, we precompute it using a new mmphf w'_c, as follows. For each node $u = child(u', a)$ in T_c, store $v = child(slink(u'), a)$ in w'_c; note each v in T_c is stored exactly once. The preorders of v follow the same order of u, and thus if we call $u' = wlink(v', c)$ (or $v' = slink(u')$), we have the desired child in $w'_c(child(v', a)) = u$.

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 (using Lemma 5) the resulting interval directly with the suffix array interval of v. 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 is sufficient. 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.

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; therefore mmphf w'_c stores at most $2n_c$ values in the range $[1, 2n]$. Therefore it

[4] 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.

requires space $O(n_c(\lg\lg \frac{n}{n_c} + 1))$ bits, which added over all $c \in [1, \sigma]$ gives a total of $O(n(\lg H_0(T) + 1))$, as in Section 3.

In order to reduce this space we partition the mmphfs according to the $O(\sigma^k)$ partitions of the BWT. Consider all the possible context strings C_i of length k,[5] their suffix tree node v_i, and their corresponding suffix array interval $A[sp_i, ep_i]$. The corresponding BWT partition is thus $S_i = T^{bwt}[sp_i, ep_i]$, of length $n_i = |S_i| = ep_i - sp_i + 1$. 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 $cC[i]$, whose number of leaves is the number of occurrences of c in S_i, $n(c, i)$. Thus the total space of all the mmphfs is $\sum_{c,i} O(n(c, i)(\lg\lg \frac{n_i}{n(c,i)} + 1)) = O(n(\lg H_k(T) + 1))$ by the log-sum inequality (recall Section 3), as $nH_k(T) = \sum_{c,i} n(c, i) \lg \frac{n_i}{n(c,i)}$.

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 $O(\sigma^k \lg n)$. It is easy, along our backward search, to know the context C_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 $nH_k(T) + o(nH_k(T)) + O(n + (n \lg n)/s + \sigma^{k+1} \lg n)$ 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 in time $t_{\mathsf{extract}} = O(s + \ell)$.*

8 Final Remarks

We have achieved alphabet independence on compressed self-indexes. This refers not only to time complexities: Even the space usage is independent of σ. The exception is the extra term $O(\sigma^{k+1} \lg n)$, but it rather limits k and it is essentially unavoidable under the k-th order empirical entropy model [12].

It is open whether we can reduce the $O(n)$ term to $o(n)$, as in the best current space result [2]. More ambitious is to achieve optimal times within optimal space, as already (partially) achieved when using $cnH_k(T)$ bits for $c > 2$ [14].

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)
2. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/Select and applications. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 315–326. Springer, Heidelberg (2010)

[5] Actually the compression booster [7] admits a more flexible partition into suffix tree nodes; we choose this way for simplicity of exposition.

3. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: SODA, pp. 680–689 (2007)
4. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In: SODA, pp. 785–794 (2009)
5. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Theory and practise of monotone minimal perfect hashing. In: ALENEX (2009)
6. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
7. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM 52(4), 688–713 (2005)
8. Ferragina, P., Grossi, R.: The string b-tree: A new data structure for string search in external memory and its applications. J. ACM 46(2), 236–280 (1999)
9. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS, pp. 390–398 (2000)
10. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
11. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)
12. Gagie, T.: Large alphabets and incompressibility. Inf. Proc. Lett. 99(6), 246–251 (2006)
13. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA, pp. 368–373 (2006)
14. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
15. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 678–689. Springer, Heidelberg (2010)
16. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: STOC, pp. 397–406 (2000)
17. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 95–106. Springer, Heidelberg (2007)
18. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
19. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
20. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
22. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA, pp. 233–242 (2002)
23. Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Lee, D.T., Teng, S.-H. (eds.) ISAAC 2000. LNCS, vol. 1969, pp. 295–321. Springer, Heidelberg (2000)
24. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Alg. 48(2), 294–313 (2003)
25. Sadakane, K.: Compressed suffix trees with full functionality. Theo. Comp. Sys. 41(4), 589–607 (2007)
26. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA, pp. 134–149 (2010)
27. Weiner, P.: Linear pattern matching algorithm. In: Proc. Ann. IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)