Compressed Full-Text Indexes

Gonzalo Navarro
Center for Web Research, Department of Computer Science, University of Chile

Veli Mäkinen
Department of Computer Science, University of Helsinki, Finland

Full-text indexes provide fast substring search over large text collections. A serious problem of these indexes has traditionally been their space consumption. A recent trend is to develop indexes that exploit the compressibility of the text, so that their size is a function of the compressed text length. This concept has evolved into *self-indexes*, which in addition contain enough information to reproduce any text portion, so they *replace* the text. The exciting possibility of an index that takes space close to that of the compressed text, replaces it, and in addition provides fast search over it, has triggered a wealth of activity and produced surprising results in a very short time, which radically changed the status of this area in less than five years. The most successful indexes nowadays are able to obtain almost optimal space and search time simultaneously.

In this paper we present the main concepts underlying (compressed) self-indexes. We explain the relationship between text entropy and regularities that show up in index structures and permit compressing them. Then we cover the most relevant self-indexes, focusing on how they exploit text compressibility to achieve compact structures that can efficiently solve various search problems. Our aim is to give the background to understand and follow the developments in this area.

Categories and Subject Descriptors: E.1 [**Data structures**]; E.2 [**Data storage representations**]; E.4 [**Coding and information theory**]: Data compaction and compression; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*; H.2.1 [**Database management**]: Physical design—*Access methods*; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Text indexing, text compression, entropy.

First author funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. Second author funded by the Academy of Finland under grant 108219.
Address: Gonzalo Navarro, Blanco Encalada 2120, Santiago, Chile. E-mail: gnavarro@dcc.uchile.cl. Web: http://www.dcc.uchile.cl/∼gnavarro. Veli Mäkinen, P. O. Box 68 (Gustaf Hällströmin katu 2 b), 00014 Helsinki, Finland. Email: vmakinen@cs.helsinki.fi. Web: http://www.cs.helsinki.fi/u/vmakinen.

1. INTRODUCTION

The amount of digitally available information is growing at an exponential rate. A large part of these data consists of *text*, that is, sequences of symbols representing not only natural language, but also music, program code, signals, multimedia streams, biological sequences, time series, and so on. The amount of (just HTML) online text material in the Web was estimated, in 2002, to exceed by 30–40 times what had been printed during the whole history of mankind[1]. If we exclude strongly structured data such as relational tables, text is the medium to convey information where retrieval by content is best understood. The recent boom on XML advocates the use of text as the medium to express structured and semistructured data as well, making text the favorite format for information storage, exchange, and retrieval.

Each scenario where text is used to express information requires a different form of retrieving such information from the text. There is a basic search task, however, that underlies all those applications. *String matching* is the process of finding the occurrences of a short string (called the *pattern*) inside a (usually much longer) string called the *text*. Virtually every text-managing application builds on basic string matching to implement more sophisticated functionalities such as finding frequent words (in natural language texts for Information Retrieval tasks) or retrieving sequences similar to a sample (in a gene or protein database for Computational Biology applications). Significant developments in basic string matching have a wide impact on most applications. This is our focus.

String matching can be carried out in two forms. *Sequential* string matching requires no preprocessing of the text, but rather traverses it sequentially to point out every occurrence of the pattern. *Indexed* string matching builds a data structure (*index*) on the text beforehand, which permits finding all the occurrences of any pattern without traversing the whole text. Indexing is the choice when (*i*) the text is so large that a sequential scanning is prohibitively costly, (*ii*) the text does not change so frequently that the cost of building and maintaining the index outweighs the savings on searches, and (*iii*) there is sufficient *storage space* to maintain the index and provide efficient access to it.

While the first two considerations refer to the convenience of indexing compared to sequentially scanning the text, the last one is a necessary condition to consider indexing at all. At first sight, the storage issue might not seem significant given the common availability of massive storage. The real problem, however, is efficient access. In the last two decades, CPU speeds have been doubling every 18 months, while disk access times have stayed basically unchanged. CPU caches are many times faster than standard main memories. On the other hand, the classical indexes for string matching require from 4 to 20 times the text size [McCreight 1976; Manber and Myers 1993; Kurtz 1998]. This means that, even when we may have enough main memory to hold a text, we may need to use the disk to store the index.

[1] In the UC Berkeley report *How Much Information?*, `http://www.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm`, they estimate the size of the deep Web (i.e., including public static and dynamic pages) to be between 66,800 and 91,850 terabytes, and that 17.8% of that is in text (HTML) form. In the JEP white paper *The Deep Web: Surfacing Hidden Value* by M. Bergman (U. of Michigan Press), `http://www.press.umich.edu/jep/07-01/bergman.html`, they estimate the whole printed material along written history is 390 terabytes.

Moreover, most existing indexes are not designed to work in secondary memory, so using them from disk is extremely inefficient [Ferragina and Grossi 1999]. As a result, indexes are usually confined to the case where the text is so small that even the index fits in main memory, and those cases are less interesting for indexing given consideration (i): For such a small text, a sequential scanning can be preferable for its simplicity and better cache usage compared to an indexed search.

Text compression is a technique to represent a text using less space. Given the relation between main and secondary memory access times, it is advantageous to store a large text that does not fit in main memory in compressed form, so as to reduce disk transfer time, and then decompress it by chunks in main memory prior to sequential searching. Moreover, a text may fit in main memory once compressed, so compression may completely remove the need to access the disk. Some developments in recent years have focused on improving this even more by directly searching the compressed text instead of decompressing it.

Several attempts to reduce the space requirements of text indexes were made in the past with moderate success, and some of them even considered the relation with text compression. Three important concepts have emerged.

Definition 1 *A* succinct index *is an index that provides fast search functionality using a space proportional to that of the text itself (say, two times the text size). A stronger concept is that of a* compressed index*, which takes advantage of the regularities of the text to operate in space proportional to that of the compressed text. An even more powerful concept is that of a* self-index*, which is a compressed index that, in addition to providing search functionality, contains enough information to efficiently reproduce any text substring. A self-index can therefore* replace *the text.*

Classical indexes such as suffix trees and arrays are not succinct. On a text of n characters over an alphabet of size σ, those indexes require $\Theta(n \log n)$ bits of space, whereas the text requires $n \log \sigma$ bits. The first succinct index we know of is by Kärkkäinen and Ukkonen [1996a]. It uses Lempel-Ziv compression concepts to achieve $O(n \log \sigma)$ bits of space. Indeed, this is a compressed index achieving space proportional to the k-th order entropy of the text (a lower-bound estimate for the compression achievable by many compressor families). However, it was not until this decade that the first self-index appeared [Ferragina and Manzini 2000] and the potential of the relationship between text compression and text indexing was fully realized, in particular regarding the correspondence between the entropy of a text and the regularities arising in some widely used indexing data structures. Several other succinct and self-indexes appeared almost simultaneously [Mäkinen 2000; Grossi and Vitter 2000; Sadakane 2000]. The fascinating concept of a self-index that requires space close to that of the compressed text, provides fast searching on it, and moreover replaces the text, has triggered much interest on this issue and produced surprising results in very few years.

At this point, there exist indexes that require space close to that of the best existing compression techniques, and provide search and text recovering functionality with almost optimal time complexity [Ferragina and Manzini 2005; Grossi et al. 2003; Ferragina et al. 2006]. More sophisticated problems are also starting to receive attention. For example, there are studies on efficient construction in little space [Hon et al. 2003], management of secondary storage [Mäkinen et al. 2004], searching for more complex patterns [Huynh et al. 2006], updating upon text changes [Hon et al. 2004], and so on. Furthermore, the implementation and practical aspects of the indexes is becoming a focus of attention. In particular, we point out the existence of *PizzaChili*, a repository of standardized implementations of succinct full-text indexes and testbeds, freely available at mirrors `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`. Overall, this is an extremely exciting research area, with encouraging results of theoretical and practical interest, and a long way ahead.

The aim of this survey is to give the theoretical background needed to understand and follow the developments in this area. We first give the reader a superficial overview of the main intuitive ideas behind the main results. This is sufficient to have a rough image of the area, but not to master it. Then we begin the more technical exposition. We explain the relationship between the compressibility of a text and the regularities that show up in its indexing structures. Next, we cover the most relevant existing self-indexes, focusing on how they exploit the regularities of the indexes to compress them and still efficiently handle them.

We do not give experimental results in this paper. Doing this seriously and thoroughly is a separate project. Some comparisons can be found, for example, by Mäkinen and Navarro [2005a]. A complete series of experiments is planned on the *PizzaChili* site, and the results should appear in there soon.

Finally, we aim at indexes that work for general texts. Thus we do not cover the very well-known *inverted indexes*, which only permit word and phrase queries on *natural language* texts. Natural language not only excludes symbol sequences of interest in many applications, such as DNA, gene or protein sequences in computational biology; MIDI, audio, and other multimedia signals; source and binary program code; numeric sequences of diverse kinds; etc. It also excludes many important human languages! In this context, *natural language* refers only to languages where words can be syntactically separated and follow some statistical laws [Baeza-Yates and Ribeiro 1999]. This encompasses English and several other European languages, but it excludes, for example, Chinese and Korean, where words are hard to separate without understanding the meaning of the text. It also excludes agglutinating languages such as Finnish and German, where "words" are actually concatenations of the particles one wishes to search.

When applicable, inverted indexes require only 20%–100% of extra space on top of the text [Baeza-Yates and Ribeiro 1999]. Moreover, there exist compression techniques that can represent inverted index and text in about 35% of the space required by the original text [Witten et al. 1999; Navarro et al. 2000; Ziviani et al. 2000], yet those indexes only point to the documents where the query words appear.

2. NOTATION AND BASIC CONCEPTS

A *string* S is a sequence of *characters*. Each character is an element of a finite set called *alphabet*. The alphabet is usually called Σ and its size $|\Sigma| = \sigma$, and it is assumed to be totally ordered. Sometimes we assume $\Sigma = [1, \sigma] = \{1, 2, \ldots, \sigma\}$. The *length* (number of characters) of S is denoted $|S|$. Let $n = |S|$, then the characters of S are indexed from 1 to n, so that the i-th character of S is S_i or $S[i]$.

A *substring* of S is written $S_{i,j} = S_i S_{i+1} \ldots S_j$. A *prefix* of S is a substring of the form $S_{1,j}$, and a *suffix* is a substring of the form $S_{i,n}$. If $i > j$ then $S_{i,j} = \varepsilon$, the empty string of length $|\varepsilon| = 0$.

The *concatenation* of strings S and S', written SS', is obtained by appending S' at the end of S. It is also possible to concatenate string S and character c, as in cS or Sc. By c^i we denote i concatenated repetitions of c: $c^0 = \varepsilon$, $c^{i+1} = c^i c$.

The *lexicographical order* "$<$" among strings is defined as follows. Let a and b be characters and X and Y be strings. Then $aX < bY$ if $a < b$, or if $a = b$ and $X < Y$. Furthermore, $\varepsilon < X$ for any $X \neq \varepsilon$.

The problems we focus on in this paper are defined as follows.

Definition 2 *Given a (long)* text *string $T_{1,n}$ and a (comparatively short)* pattern *string $P_{1,m}$, both over alphabet Σ, the* occurrence positions *(or just* occurrences*) of P in T are the set $O = \{1 + |X|,\ \exists Y,\ T = XPY\}$. Two* search problems *are of interest: (1)* count *the number of occurrences, that is, return occ $= |O|$; (2)* locate *the occurrences, that is, return set O in some order. When the text T is not explicitly available, a third task of interest is (3)* display *text substrings, that is, return $T_{l,r}$ given l and r.*

In this paper we adopt for technical convenience the assumption that T is terminated by $T_n = \$$, which is a character from Σ that lexicographically precedes all the others and appears nowhere else in T nor in P.

Logarithms in this paper are in base 2 unless otherwise stated.

In our study of compression algorithms, we will need routines to access individual bit positions inside bit vectors. This raises the question of which machine model to assume. We assume the standard *word* random access model (RAM); the *computer word size* w is assumed to be such that $\log n = O(w)$, where n is the maximum size of the problem instance. Standard operations (like bit-shifts, additions, etc.) on an $O(w) = O(\log n)$-bit integer are assumed to take constant time in this model. However, all the results considered in this paper only assume that an $O(w)$-bit block at any given position in a bit vector can be read, written, and converted into an integer, in constant time. This means that on a weaker model, where for example such operations would take time linear in the length of the bit block, all the time complexities for the basic operations should be multiplied by $O(\log n)$.

A table of the main symbols, with short explanations and pointers to their definitions, is given in Appendix A.

3. BASIC TEXT INDEXES

Given the focus of this paper, we are not covering the various text indexes that have been designed with constant-factor space reductions in mind, with no relation to text compression nor self-indexing. In general these indexes have had some, but not spectacular, success in lowering the large space requirements of text indexes [Blumer et al. 1987; Andersson and Nilsson 1995; Kärkkäinen 1995; Irving 1995; Colussi and de Col 1996; Kärkkäinen and Ukkonen 1996b; Crochemore and Vérin 1997; Kurtz 1998; Giegerich et al. 2003].

In this section, in particular, we introduce the most classical full-text indexes, which are those that are turned into compressed indexes later in this paper.

3.1 Tries or Digital Trees

A digital tree or *trie* [Fredkin 1960; Knuth 1973] is a data structure that stores a set of strings. It can support the search for a string in the set in time proportional to the length of the string sought, independently of the set size.

Definition 3 *A* trie *for a set $\mathcal{S}$ of distinct strings $S^1,\ S^2 \ldots, S^N$ is a tree where each node represents a distinct prefix in the set. The root node represents the empty prefix ε. Node v representing prefix Y is a child of node u representing prefix X iff $Y = Xc$ for some character c, which will label the tree edge from u to v.*

We assume that all strings are terminated by "\$". Under this assumption, no string S^i is a prefix of another, and thus the trie has exactly N leaves, each corresponding to a distinct string. Fig. 1 illustrates.

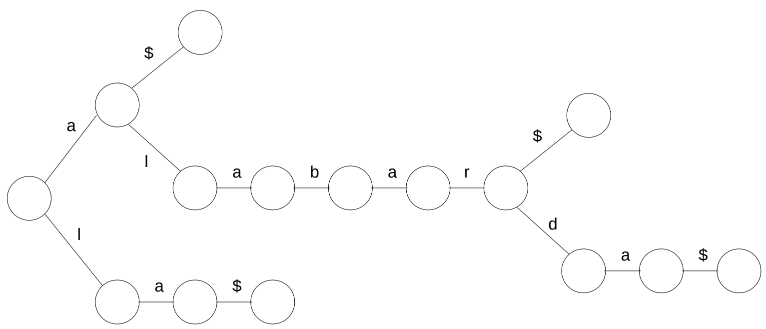

Fig. 1. A trie for the set {`"alabar"`, `"a"`, `"la"`, `"alabarda"`}. In general the arity of trie nodes may be as large as the alphabet size.

A trie for $\mathcal{S} = \{S^1,\ S^2, \ldots, S^N\}$ is easily built in time $O(|S^1|+|S^2|+\ldots+|S^N|)$ by successive insertions. Any string S can be searched for in the trie in time $O(|S|)$ by following from the trie root the path labeled with the characters of S. Two outcomes are possible: (i) at some point i there is no edge labeled S_i to follow, which means that S is not in the set $\mathcal{S}$, (ii) we reach a leaf corresponding to S (assume that S is also terminated with character "\$").

Actually, the above complexities assume that the alphabet size σ is a constant. For general σ, we must multiply the above complexities by $O(\log \sigma)$, which accounts for the overhead of searching the correct character to follow inside each node. This can be made $O(1)$ by using at each node a direct addressing table of size σ, but in this case the size and construction cost must be multiplied by $O(\sigma)$ to allocate the tables at each node. Alternatively, perfect hashing of the children of each node permits $O(1)$ search time and $O(1)$ space factor, yet the construction cost is multiplied by $O(\sigma^2)$ [Raman 1996].

Note that a trie can be used for *prefix searching*, that is, to find every string prefixed by S in the set (in this case, S is not terminated with "\$"). If we can follow the trie path corresponding to S, then the internal node reached corresponds to all the strings S^i prefixed by S in the set. We can then traverse all the leaves of the subtree to find the answers.

3.2 Suffix Tries and Suffix Trees

Let us now consider how tries can be used for text indexing. Given text $T_{1,n}$ (terminated with $T_n = \$$), T defines n suffixes $T_{1,n},\ T_{2,n}, \ldots, T_{n,n}$.

Definition 4 *The* suffix trie *of a text T is a trie data structure built over all the suffixes of T.*

The suffix trie of T makes up an index for fast string matching. Given a pattern $P_{1,m}$ (not terminated with "\$"), every occurrence of P in T is a substring of T, that is, the prefix of a suffix of T. Entering the suffix trie with the characters of P leads us to a node that corresponds to all the suffixes of T prefixed by P (or, if we do not arrive at any trie node, then P does not occur in T). This permits counting the occurrences of P in T in $O(m)$ time, by simply recording the number of leaves that descend from each suffix tree node. It also permits finding all the *occ* occurrences of P in T in $O(m + occ)$ time by traversing the whole subtree (some additional pointers threading the leaves and connecting each internal node to its first leaf are necessary to ensure this complexity).

As described, the suffix trie usually has $\Theta(n^2)$ nodes. In practice, the trie is pruned at a node as soon as there is only a unary path from the node to a leaf. Instead, a pointer to the text position where the corresponding suffix starts is stored. On average, the pruned suffix trie has only $O(n)$ nodes [Sedgewick and Flajolet 1996]. Yet, albeit unlikely, it might have $\Theta(n^2)$ nodes in the worst case. Fortunately, there exist equally powerful structures that guarantee linear space and construction time in the worst case [Morrison 1968; Apostolico 1985].

Definition 5 *The* suffix tree *of a text T is a suffix trie where each unary path is converted into a single edge. Those edges are labeled by strings obtained by concatenating the characters of the replaced path. The leaves of the suffix tree indicate the text position where the corresponding suffixes start.*

Since there are n leaves and no unary nodes, it is easy to see that suffix trees require $O(n)$ space (the strings at the edges are represented with pointers to the text). Moreover, they can be built in $O(n)$ time [Weiner 1973; McCreight 1976; Ukkonen 1995; Farach 1997]. Fig. 2 shows an example.

The search for P in the suffix tree of T is similar to a trie search. Now we may use more than one character of P to traverse an edge, but all edges leaving from a node have different first characters. The search can finish in three possible ways: (i) at some point there is no edge leaving from the current node that matches the characters that follow in P, which means that P does not occur in T; (ii) we read all the characters of P and end up at a tree node (or in the middle of an edge), in which case all the answers are in the subtree of the reached node (or edge); or (iii) we reach a leaf of the suffix tree without having read the whole P, in which case there is at most one occurrence of P in T, which must be checked by going to the suffix pointed to by the leaf and comparing the rest of P with the rest of the suffix. In any case the process takes $O(m)$ time (assuming one uses perfect hashing to find the children in constant time) and suffices for counting queries.

Suffix trees permit $O(m + occ)$ locating time without need of further pointers to thread the leaves, since the subtree with *occ* leaves has $O(occ)$ nodes. The real

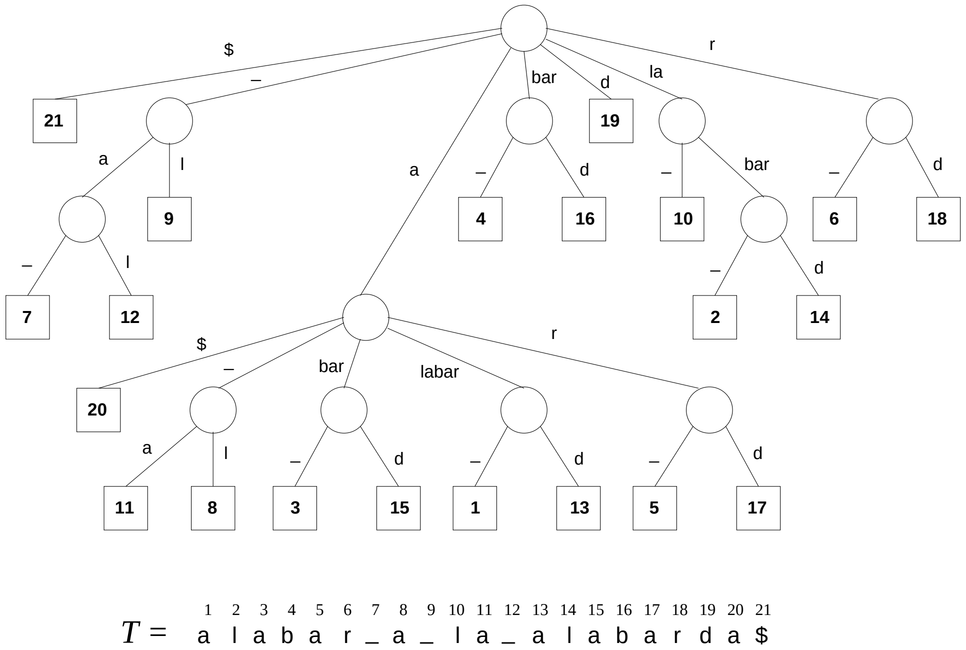

Fig. 2. The suffix tree of the text `"alabar a la alabarda$"`. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters `"a"`–`"z"`.

problem of suffix trees is their high space consumption, which is $\Theta(n \log n)$ bits and at the very least 10 times the text size in practice [Kurtz 1998].

3.3 Suffix Arrays

A *suffix array* [Manber and Myers 1993; Gonnet et al. 1992] is simply a permutation of all the suffixes of T so that the suffixes are lexicographically sorted.

Definition 6 *The* suffix array *of a text $T_{1,n}$ is an array $A[1,n]$ containing a permutation of the interval $[1,n]$, such that $T_{A[i],n} < T_{A[i+1],n}$ for all $1 \le i < n$, where "$<$" between strings is the lexicographical order.*

The suffix array can be obtained by collecting the leaves of the suffix tree in left-to-right order (assuming that the children of the suffix tree nodes are lexicographically ordered left-to-right by the edge labels). However, it is much more practical to build them directly. In principle, any comparison-based sorting algorithm can be used, as it is a matter of sorting the n suffixes of the text, but this could be costly especially if there are long repeated substrings within the text. There are several more sophisticated algorithms, from the original $O(n \log n)$ time [Manber and Myers 1993] to the latest $O(n)$ time algorithms [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen and Sanders 2003]. In practice, the best current algorithms are not linear-time ones [Larsson and Sadakane 1999; Itoh and Tanaka 1999; Manzini and Ferragina 2004; Schürmann and Stoye 2005]. See a comprehensive survey in [Puglisi et al. 2006].

Fig. 3 shows our example suffix array. Note that each subtree of the suffix tree corresponds to the suffix array subinterval encompassing all its leaves (in the figure we have shaded the interval corresponding to the suffix tree node representing `"a"`). Note also that, since $T_n = \$$, we always have $A[1] = n$, as $T_{n,n}$ is the smallest suffix.

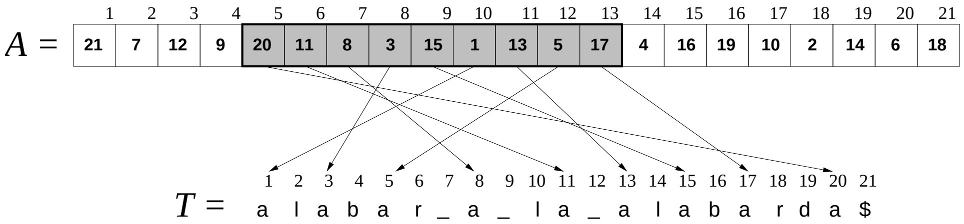

Fig. 3. The suffix array of the text `"alabar a la alabarda$"`. We have shown explicitly where the suffixes starting with `"a"` point to.

The suffix array plus the text contain enough information to search for patterns. Since the result of a suffix tree search is a subtree, the result of a suffix array search must be an interval. This is also obvious if one considers that all the suffixes prefixed by P are lexicographically contiguous in the sorted array A. Thus, it is possible to search for the interval of A containing the suffixes prefixed by P via two binary searches on A. The first binary search determines the starting position sp for the suffixes lexicographically larger than or equal to P. The second binary search determines the ending position ep for suffixes that start with P. Then the answer is the interval $A[sp, ep]$. A counting query needs to report just $ep - sp + 1$. A locating query enumerates $A[sp]$, $A[sp + 1]$, ..., $A[ep]$.

Note that each step of the binary searches requires a lexicographical comparison between P and some suffix $T_{A[i],n}$, which requires $O(m)$ time in the worst case. Hence the search takes worst case time $O(m \log n)$ (this can be lowered to $O(m + \log n)$ by using more space to store the length of the longest common prefixes between consecutive suffixes [Manber and Myers 1993; Abouelhoda et al. 2004]). A locating query requires additional $O(occ)$ time to report the occ occurrences. Alg. 1 gives the pseudocode.

Algorithm SASearch($P_{1,m}$,$A[1,n]$,$T_{1,n}$)

$sp \leftarrow 1$; $st \leftarrow n + 1$;
while $sp < st$ **do**
    $s \leftarrow \lfloor (sp + st)/2 \rfloor$;
    if $P > T_{A[s],A[s]+m-1}$ **then** $sp \leftarrow s + 1$ **else** $st \leftarrow s$;
$ep \leftarrow sp - 1$; $et \leftarrow n$;
while $ep < et$ **do**
    $e \leftarrow \lceil (ep + et)/2 \rceil$;
    if $P = T_{A[e],A[e]+m-1}$ **then** $ep \leftarrow e$ **else** $et \leftarrow e - 1$;
return $[sp, ep]$;

Alg. 1: Searching for P in the suffix array A of text T. T is assumed to be terminated by "\$", but P is not. Accesses to T outside the range $[1, n]$ are assumed to return "\$". From the returned data one can answer the counting query $ep - sp + 1$ or the locating query $A[sp, ep]$.

All the space/time tradeoffs offered by the different compressed full-text indexes in this paper will be expressed in tabular form, as theorems where the meaning of n, m, σ, and H_k, is implicit (see Section 5 for the definition of H_k). For illustration and comparison, and because suffix arrays are the main focus of compressed indexes, we give now the corresponding theorem for suffix arrays. As the suffix array is not a self-index, the space in bits includes a final term $n \log \sigma$ for the text itself. The time to count refers to counting the occurrences of $P_{1,m}$ in $T_{1,n}$. The time to locate refers to giving the text position of a single occurrence after counting has completed. The time to display refers to displaying one contiguous text substring of ℓ characters. In this case (not a self-index), this is trivial as the text is readily available. We show two tradeoffs, the second referring to storing longest common prefix information. Throughout the survey, many tradeoffs will be possible for the structures we review, and we will choose to show only those that we judge most interesting.

Theorem 1 [Manber and Myers 1993] *The* Suffix Array *(*SA*) offers the following space/time tradeoffs.*

Space in bits	$n \log n + n \log \sigma$
Time to count	$O(m \log n)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$
Space in bits	$2n \log n + n \log \sigma$
Time to count	$O(m + \log n)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$

4. PRELUDE TO COMPRESSED FULL-TEXT INDEXING

Before we start the formal and systematic exposition of the techniques that lead to compressed indexing, we want to point out some key ideas at an informal level. This is to permit the reader to understand the essential concepts without thoroughly absorbing the formal treatment that follows. The section also includes a "roadmap" guide for the reader to gather from the forthcoming sections the details required to fully grasp what is behind the simplified presentation of this section.

We explain two basic concepts that play a significant role in compressed full-text indexing. Interestingly, these two concepts can be plugged as such to traditional full-text indexes to achieve immediate results. No knowledge of compression techniques is required to understand the power of these methods. These are *backward search* [Ferragina and Manzini 2000] and *wavelet trees* [Grossi et al. 2003].

After introducing these concepts, we will give a brief motivation to compressed data structures by showing how a variant of the familiar *inverted index* can be easily turned into a compressed index for natural language texts. Then we explain how this same compression technique can be used to implement an approach that is the reverse of backward searching.

4.1 Backward Search

Recall the binary search algorithm in Alg. 1. Ferragina and Manzini [2000] propose a completely different way of guiding the search: The pattern is searched for from its last to its first character. Fig. 4 illustrates how this backward search proceeds.

Backward step 1: search for "a"

i	A[i]	$T_{A[i]-1}$	suffix $T_{A[i],n}$
1:	21	a	$
2:	7	r	_a_la_alabarda$
3:	12	a	_alabarda$
4:	9	a	_la_alabarda$
5:	20	d	a$
6:	11	l	a_alabarda$
7:	8	_	a_la_alabarda$
8:	3	l	abar_a_la_alabarda$
9:	15	l	abarda$
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$
12:	5	b	ar_a_la_alabarda$
13:	17	b	arda$
14:	4	a	bar_a_la_alabarda$
15:	16	a	barda$
16:	19	r	da$
17:	10	_	la_alabarda$
18:	2	a	labar_a_la_alabarda$
19:	14	a	labarda$
20:	6	a	r_a_la_alabarda$
21:	18	a	rda$

Backward step 2: search for "la"

i	A[i]	$T_{A[i]-1}$	suffix $T_{A[i],n}$
1:	21	a	$
2:	7	r	_a_la_alabarda$
3:	12	a	_alabarda$
4:	9	a	_la_alabarda$
5:	20	d	a$
6:	11	l	a_alabarda$
7:	8	_	a_la_alabarda$
8:	3	l	abar_a_la_alabarda$
9:	15	l	abarda$
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$
12:	5	b	ar_a_la_alabarda$
13:	17	b	arda$
14:	4	a	bar_a_la_alabarda$
15:	16	a	barda$
16:	19	r	da$
17:	10	_	la_alabarda$
18:	2	a	labar_a_la_alabarda$
19:	14	a	labarda$
20:	6	a	r_a_la_alabarda$
21:	18	a	rda$

Backward step 3: search for "ala"

i	A[i]	suffix $T_{A[i],n}$
1:	21	$
2:	7	_a_la_alabarda$
3:	12	_alabarda$
4:	9	_la_alabarda$
5:	20	a$
6:	11	a_alabarda$
7:	8	a_la_alabarda$
8:	3	abar_a_la_alabarda$
9:	15	abarda$
10:	1	alabar_a_la_alabarda$
11:	13	alabarda$
12:	5	ar_a_la_alabarda$
13:	17	arda$
14:	4	bar_a_la_alabarda$
15:	16	barda$
16:	19	da$
17:	10	la_alabarda$
18:	2	labar_a_la_alabarda$
19:	14	labarda$
20:	6	r_a_la_alabarda$
21:	18	rda$

Fig. 4. Backward search for pattern `"ala"` on the suffix array of the text `"alabar a la alabarda$"`. Here A is drawn vertically and the suffixes are shown explicitly, compare to Fig. 3.

Fig. 4 shows the steps a backward search algorithm takes when searching for the occurrences of `"ala"` in `"alabar a la alabarda$"`. Let us reverse engineer how the algorithm works. The first step is finding the range $A[5, 13]$ where the suffixes start with `"a"`. This is easy: All one needs is an array C indexed by the characters, such that $C[\texttt{"a"}]$ tells how many characters in T are smaller than `"a"` (that is, the $C[\texttt{"a"}] + 1$ points to the first index in A where the suffixes start with `"a"`). Then, knowing that `"b"` is the successor of `"a"` in the alphabet, the last index in A where the suffixes start with `"a"` is $C[\texttt{"b"}]$.

To understand step 2 in Fig. 4, consider the column labeled $T_{A[i]-1}$. By concatenating a character in this column with the suffix $T_{A[i],n}$ following it, one obtains suffix $T_{A[i]-1,n}$. Since we have found out that suffixes $T_{A[5],21}, T_{A[6],21}, \ldots, T_{A[13],21}$ are the *only ones* starting with `"a"`, we know that suffixes $T_{A[5]-1,21}$, $T_{A[6]-1,21}$, $\ldots$, $T_{A[13]-1,21}$ are the *only candidates* to start with `"la"`. We just need to check which of those candidates actually start with `"l"`.

The crux is to efficiently find the range corresponding to suffixes starting with `"la"` after knowing the range corresponding to `"a"`. Consider all the concatenated suffixes `"l"`$T_{A[i],n}$ in the descending row order they appear in (that is, find the rows i where $T_{A[i]-1} = \texttt{"l"}$). Now find the rows i' for the corresponding suffixes $T_{A[i'],n} = T_{A[i]-1,n} = \texttt{"l"}T_{A[i],n}$ (note that $A[i'] = A[i] - 1$): Row $i = 6$ becomes row $i' = 17$ after we prepend `"l"` to the suffix, row 8 becomes row 18, and row 9 becomes row 19. One notices that the top-to-bottom order in the suffix array is preserved! It is easy to see why this must be so: the suffixes $T_{A[i]-1,n}$ that start with "l" must be sorted according to the characters that follow that "l", and this is precisely how suffixes $T_{A[i],n}$ are sorted.

Hence, to discover the new range corresponding to "la" it is sufficient to count how many times "l" appears before row 5 and up to row 13 in column $T_{A[i]-1}$. The counts are 0 and 3, and hence the complete range $[17, 19]$ of suffixes starting with "l" (found with C) also start with "la".

Step 3 is more illustrating. Following the same line of thought, we end up counting how many times the first character of our query, "a", appears before row 17 and up to row 19 in column $T_{A[i]-1}$. The counts are 5 and 7. This means that the range corresponding to suffixes starting with "ala" is $[C[\texttt{"a"}]+1+5, C[\texttt{"a"}]+7] = [10, 11]$.

We have now discovered that backward search can be implemented by means of a small table C and some queries on the column $T_{A[i]-1}$. Let us denote this column by string $L_{1,n}$ (for reasons to be made clear in Section 5.3). One notices that the *single* query needed on L is counting how many times a given character c appears up to some given position i. Let us denote this query $Occ(c, i)$. For example, in Fig. 4 we have $L =$"araadl_ll$_bbaar_aaaa" and $Occ(\texttt{"a"}, 16) = 5$ and $Occ(\texttt{"a"}, 19) = 7$.

Note that we can define a function $LF(i) = C[L_i] + Occ(L_i, i)$, so that if $LF(i) = i'$, then $A[i'] = A[i] - 1$. This permits decoding the text backwards, mapping suffix $T_{A[i],n}$ to $T_{A[i]-1,n}$. Instead of mapping one suffix at a time, the backward search maps a range of suffixes (those prefixed by the current pattern suffix $P_{i...m}$) to their predecessors having the required first character P_{i-1} (so the suffixes in the new range have the common prefix $P_{i-1...m}$). Section 9 gives more details.

To finish the description of backward search, we still have to discuss how the function $Occ(c, i)$ can be computed. The most naive way to solve the $Occ(c, i)$ query is to do the counting on each query. However, this means $O(n)$ scanning at each step (overall $O(mn)$ time!). Another extreme is to store all the answers in an array $Occ[c, i]$. This requires $\sigma n \log n$ bits of space, but gives $O(m)$ counting time, which improves the original suffix array search complexity. A practical implementation of backward search is somewhere in between the extremes: Consider *indicator* bit vectors $B^c[i] = 1$ iff $L_i = c$ for each character c. Let us define operation $rank_b(B, i)$ as the number of occurrences of bit b in $B[1, i]$. It is easy to see that $rank_1(B^c, i) = Occ(c, i)$. That is, we have reduced the problem of counting characters up to a given position in string L to counting bits set up to a given position in bit vectors. Function *rank* will be studied in Section 6, where it will be shown that some simple dictionaries taking $o(n)$ extra bits for a bit vector B of length n enable answering $rank_b(B, i)$ in constant time for any i. By building these dictionaries for the indicator bit-vectors B^c, we can conclude that $\sigma n + o(\sigma n)$ bits of space suffices for $O(m)$ time backward search. These structures, together with the basic suffix array, give the following result.

Theorem 2 *The* Suffix Array with *rank*-dictionaries *(*SA-R*) supports backward search with the following space and time complexities.*

Space in bits	$n \log n + \sigma n + o(\sigma n)$
Time to count	$O(m)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$

4.2 Wavelet trees

A tool to reduce the alphabet dependence from σn to $n \log \sigma$ in the space to solve $Occ(c, i)$ is the *wavelet tree* of Grossi, Gupta, and Vitter [2003]. The idea is to simulate each $Occ(c, i)$ query by $\log \sigma$ *rank*-queries on binary sequences. See Fig. 5.

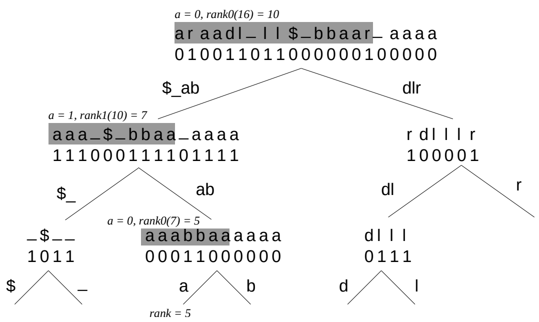

Fig. 5. A binary wavelet tree for the string $L =$ `"araadl_ll$_bbaar_aaaa"`, illustrating the solution of query $Occ($`"a"`$, 15)$. Only the bit vectors are stored, the texts are shown for clarity.

The wavelet tree is a balanced search tree where each symbol from the alphabet corresponds to a leaf. The root holds a bit vector marking with 1 those positions whose corresponding characters descend to the right. Those characters are concatenated to form the sequence corresponding to the right child of the root. The characters at positions marked 0 in the root bit vector make up the sequence corresponding to the left child. The process is repeated recursively until the leaves. Only the bit vectors marking the positions are stored, and they are preprocessed for *rank*-queries.

Fig. 5 shows how $Occ($`"a"`$, 16)$ is computed in our example. As we know that `"a"` belongs to the first half of the sorted alphabet[2], it receives mark 0 in the root bit vector B, and consequently its occurrences go to the left child. Thus, we compute $rank_0(B, 16) = 10$ to find out which is its corresponding character position in the sequence of the left child of the root. As `"a"` belongs to the second quarter of the sorted alphabet (that is, to the second half within the first half), its occurrences are marked 1 in the bit vector B' of the left child of the root. Thus we compute $rank_1(B', 10) = 7$ to find out the corresponding position in the right child of the current node. The third step computes $rank_0(B'', 7) = 5$ in that node, so we arrive at a leaf with position 5. That leaf would contain a sequence formed by just `"a"`s, thus we already have our answer $Occ($`"a"`$, 16) = 5$. (The process is similar to *fractional cascading* in computational geometry [de Berg et al. 2000, Chapter 5].)

With some care (see Section 6.3) the wavelet tree can be represented in $n \log \sigma + o(n \log \sigma)$ bits, supporting the internal *rank* computations in constant time. As we have seen, each $Occ(c, i)$ query can be simulated by $\log \sigma$ binary *rank* computations. That is, wavelet trees enable improving the space complexity significantly with a small sacrifice in time complexity.

[2]In practice, this can be done very easily: one can use the bits of the integer representation of the character within Σ, from most to least significant.

Theorem 3 *The* Suffix Array with wavelet trees *(*SA-WT*) supports backward search with the following space and time complexities.*

Space in bits	$n \log n + 2n \log \sigma + o(n \log \sigma)$
Time to count	$O(m \log \sigma)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$

4.3 Turning Suffix Arrays into Self-Indexes

We have seen that, using wavelet trees, counting queries can actually be supported without using suffix arrays at all. For locating the occurrence positions or displaying text context, the suffix array and original text are still necessary. The main technique to cope without them is to sample the suffix array at regular text position intervals (that is, given a sampling step b, collect all entries $A[i] = b \cdot j$ for every j).

Then, to locate the occurrence positions we proceed as follows: The counting query gives an interval in the suffix array to be reported. Now, given each position i within this interval, we wish to find the corresponding text position $A[i]$. If suffix position i is not sampled, one performs a backward step $LF(i)$ to find the suffix array entry pointing to text position $A[i]-1$, $A[i]-2$, $\ldots$, until a sampled position $X = A[i] - k$ is found. Then, $X + k$ is the answer. Displaying arbitrary text substrings $T_{l,r}$ is also easy by first finding the nearest sampled position after r, $A[i] = r' > r$, and then using LF repeatedly over i until traversing all the positions backward until l. The same wavelet tree can be used to reveal the text characters to display between positions r and l, each in $\log \sigma$ steps. More complete descriptions are given in the next sections (small details vary among indexes).

It should be clear that the choice of the sampling step involves a tradeoff between extra space for the index and time to locate occurrences and display text substrings.

This completes the description of a simple self-index: we do not need the text nor the suffix array, just the wavelet tree and a few additional arrays. This index is not yet compressed. Compression can be obtained, for example, by using more compact representations of wavelet trees (see Sections 6.3 and 9).

4.4 Forward Searching: Compressed Suffix Arrays

Another line of research on self-indexes [Grossi and Vitter 2000; Sadakane 2000] builds on the *inverse* of function $LF(i)$. This inverse function, denoted Ψ, maps suffix $T_{A[i],n}$ to suffix $T_{A[i]+1,n}$, and thus it enables scanning the text in forward direction, from left to right. Continuing our example, we computed $LF(6) = 17$, and thus the inverse is $\Psi(17) = 6$. Mapping Ψ has the simple definition $\Psi(i) = i'$ such that $A[i'] = (A[i] \bmod n) + 1$. Fig. 6 illustrates.

While the $Occ(c, i)$ function demonstrated the backward search paradigm, the inverse function Ψ is useful in demonstrating the connection to compression: when its values are listed from 1 to n, they form σ increasing integer sequences with each value in $\{1, 2, \ldots, n\}$. Fig. 6 works as a proof by example. Such increasing integer sequences can be compressed using so-called *gap encoding* methods, as will be demonstrated in Section 4.5.

Let us, for now, assume that we have the Ψ values compressed in a form that enables constant-time access to its values. This function and the same table C that

i	A[i]	Ψ	suffix $T_{A[i],n}$
1:	21	10	$
2:	7	7	_a_la_alabarda$
3:	12	11	_alabarda$
4:	9	17	_la_alabarda$
5:	20	1	a$
6:	11	3	a_alabarda$
7:	8	4	a_la_alabarda$
8:	3	14	abar_a_la_alabarda$
9:	15	15	abarda$
10:	1	18	alabar_a_la_alabarda$
11:	13	19	alabarda$
12:	5	20	ar_a_la_alabarda$
13:	17	21	arda$
14:	4	12	bar_a_la_alabarda$
15:	16	13	barda$
16:	19	5	da$
17:	10	6	la_alabarda$
18:	2	8	labar_a_la_alabarda$
19:	14	9	labarda$
20:	6	2	r_a_la_alabarda$
21:	18	16	rda$

Fig. 6. The suffix array of the text `"alabar a la alabarda$"` with Ψ values computed.

was used in backward searching are almost all we need. Consider the standard binary search algorithm of Alg. 1. Each step requires comparing a prefix of some text suffix $T_{A[i],n}$ with the pattern. We can extract such a prefix by following the Ψ values recursively: i, $\Psi[i]$, $\Psi[\Psi[i]]$, $\ldots$, as they point to $T_{A[i]}$, $T_{A[i]+1}$, $T_{A[i]+2}$, $\ldots$ After at most m steps, we have revealed enough characters from the suffix the pattern is being compared to. To discover each such character $T_{A[j]}$, for $j = i$, $\Psi[i]$, $\Psi[\Psi[i]]$, $\ldots$, we can use the same table C: because the first characters $T_{A[j]}$ of the suffixes $T_{A[j],n}$, are in alphabetic order in A, $T_{A[j]}$ must be the character c such that $C[c] < j \le C[c+1]$. Thus each $T_{A[j]}$ can be found by an $O(\log \sigma)$-time binary search on C. By spending $n + o(n)$ extra bits, this binary search on C can be replaced by a constant-time *rank* (details in Section 6.1). Let $\mathtt{newF}_{1,n}$ be a bit vector where we mark the positions in A where suffixes change their first character, and assume for simplicity that all the characters of Σ appear in T. In our example, $\mathtt{newF} = 110010000000010110010$. Then, the character pointed from $A[j]$ is $rank_1(\mathtt{newF}, j)$.

The overall time complexity for counting queries is therefore the same $O(m \log n)$ as in the standard binary search. For locating and displaying the text, one can use the same sampling method of Section 4.3. We give more details in Section 8.

4.5 Compressing Ψ and Inverted Indexes

The same technique used for compressing function Ψ is widely used in the more familiar *inverted indexes* for natural language texts. We analyze the space usage of inverted indexes as an introduction to compression of function Ψ.

Consider the text `"to be or not to be"`. An inverted index for this text is

`"be"`:	$4, 17$
`"not"`:	10
`"or"`:	7
`"to"`:	$1, 14$

That is, each word in the vocabulary is followed by its occurrence positions in the text, in increasing order. It is then easy to search for the occurrences of a query word: find it in the vocabulary (by binary search, or using a trie on the vocabulary words) and fetch its list of occurrences. We can think of the alphabet of the text as $\Sigma = \{\texttt{"be"},\texttt{"not"},\texttt{"or"},\texttt{"to"}\}$. Then we have $\sigma = |\Sigma|$ increasing occurrence lists to compress (exactly as we will have in function Ψ later on).

An efficient way to compress the occurrence lists uses differential encoding plus a variable-length coding, such as Elias-δ coding [Elias 1975; Witten et al. 1999]. Take the list for `"be"`: $4, 17$. First, we get smaller numbers by representing the differences (called gaps) between adjacent elements: $(4-0),(17-4) = 4, 13$. The binary representations of the numbers 4 and 13 are 100 and 1101, respectively. However, sequence 1001101 does not reveal the original content as the boundaries are not marked. Hence, one must somehow encode the lengths of the separate binary numbers. Such coding is for example $\delta(4)\delta(13) = 1101110011101001101$, where the first bits set until the first zero (11) encode a number y $(= 2)$ in unary. The next y-bit field (11), as an integer (3), tells the length of the bit field (100) that codes the actual number (4). The process is repeated for each gap. Asymptotically the code for integer x takes $\log x + 2\log\log x + O(1)$ bits, where $O(1)$ comes from the zero-bit separating the unary code from the rest, and from rounding the logarithms.

We can now analyze the space requirement of the inverted index when the differentially represented occurrence lists are δ-encoded. Let $I_w[1], I_w[2], \ldots I_w[n_w]$ be the list of occurrences for word $w \in \Sigma$. The list represents the differences between occurrence positions, hence $\sum_{i=1}^{n_w} I_w[i] \le n$. The space requirement of the inverted index is then

$$
\begin{aligned}
& \sum_{w\in\Sigma}\sum_{i=1}^{n_w}(\log I_w[i] + 2\log\log I_w[i] + O(1)) \\
\le\ & O(n) + \sum_{w\in\Sigma}\sum_{i=1}^{n_w}(\log\frac{n}{n_w} + 2\log\log\frac{n}{n_w}) \\
=\ & O(n) + n\sum_{w\in\Sigma}\frac{n_w}{n}(\log\frac{n}{n_w} + 2\log\log\frac{n}{n_w}) \\
=\ & nH_0 + O(n\log\log\sigma),
\end{aligned}
$$

where the second and the last lines follow from the properties of the logarithm function (the sum obtains its largest value when the occurrence positions are distributed regularly). We have introduced H_0 as a shorthand notation for a familiar measure of compressibility: the zero-order empirical entropy $H_0 = \sum_{w\in\Sigma}\frac{n_w}{n}\log\frac{n}{n_w}$ (see Section 5), where the text is regarded as a sequence of words. For example, nH_0 is a lower bound for the bit-length of the output file produced by any compressor that encodes each text word using a unique (variable-length) bit sequence. Those types of zero-order word-based compressors are very popular for their good results on natural language [Ziviani et al. 2000].

This kind of gap encoding is the main tool used for inverted index compression [Witten et al. 1999]. We have shown that, using this technique, the inverted index is actually a compressed index. In fact, the text is not necessary at all to carry out searches, but displaying arbitrary text substrings is not efficiently implementable using only the inverted index.

Random access to Ψ. As mentioned, the compression of function Ψ is identical to the above scheme for lists of occurrences. The major difference is that we need access to arbitrary Ψ values, not only from the beginning. A reasonably neat solution (not the most efficient possible) is to sample, say, each $\log n$-th absolute Ψ value, together with a pointer to its position in the compressed sequence of Ψ values. Access to $\Psi[i]$ is then accomplished by finding the closest absolute value, following the pointer to the compressed sequence, and uncompressing the differential values until reaching the desired entry. Value $\Psi[i]$ is then the sampled absolute value plus the sum of the differences. It is reasonably easy to uncompress each encoded difference value in constant time. There will be at most $\log n$ values to decompress, and hence any $\Psi[i]$ value can be computed in $O(\log n)$ time. The absolute samples take $O(n)$ additional bits.

With a more careful design the extraction of Ψ values can be carried out in constant time. Indexes based on function Ψ will be studied in Sections 7 and 8.

4.6 Roadmap

At this point the reader can leave with a reasonably complete and accurate intuition of the main general ideas behind compressed full-text indexing. The rest of the paper is devoted to readers seeking for a more in-depth technical understanding of the area, and thus it revisits the concepts presented in this section (as well as other omitted ones) in a more formal and systematic way.

We start in Section 5 by exposing the fundamental relationships between text compressibility and index regularities. This also reveals the ties that exist among the different approaches, proving facts that are used both in forward and backward search paradigms. The section will also introduce the fundamental concepts behind the indexing schemes that achieve higher-order compression, something we have not touched on in this section. Readers wishing to understand the algorithmic details behind compressed indexes without understanding why they achieve the promised compression bounds, can safely skip Section 5 and just accept the space complexity claims in the rest of the paper. They will have to return to this section only occasionally for some definitions.

Section 6 describes some basic compact data structures and their properties, which can also be taken for granted when reading the other sections. Thus this section can be skipped by readers wanting to understand the main algorithmic concepts of self-indexes, but not by those wishing, for example, to implement them.

Sections 7 and 8 describe the self-indexes based on forward searching using the Ψ function, and they can be read independently of Section 9, which describes the backward searching paradigm, and of Section 10, which describes Lempel-Ziv-based self-indexes (the only ones not based on suffix arrays).

The last sections finish the survey with an overview of the area and are recommended to every reader, though not essential.

5. SUFFIX ARRAY REGULARITIES AND TEXT COMPRESSIBILITY

Suffix arrays are not random permutations. When the text alphabet size σ is smaller than n, not every permutation of $[1, n]$ is the suffix array of some text (as there are more permutations than texts of length n). Moreover, the entropy of T is reflected in regularities that appear in its suffix array A. In this section we show

how some subtle kinds of suffix array regularities are related to measures of text compressibility. Those relationships are relevant later to compress suffix arrays.

The analytical results in this section are justified with intuitive arguments or informal proofs. We refer the reader to the original sources for the formal technical proofs. We sometimes deviate slightly from the original definitions, changing inessential technical details to allow for a simpler exposition.

5.1 k-th Order Empirical Entropy

Opposed to the classical notion of k-th order entropy [Bell et al. 1990], which can only be defined for infinite sources, the *k-th order empirical entropy* defined by Manzini [2001] applies to finite texts. It coincides with the statistical estimation of the entropy of a source taking the text as a finite sample of the infinite source[3]. The definition is especially useful because it can be applied to any text without resorting to assumptions on its distribution. It has become popular in the algorithmic community, for example in analyzing the size of data structures, because it is a worst-case measure but yet relates the space usage to compressibility.

Definition 7 *Let $T_{1,n}$ be a text over an alphabet Σ. The* zero-order empirical entropy *of T is defined as*

$$H_0 \;=\; H_0(T) \;=\; \sum_{c\in\Sigma, n_c>0} \frac{n_c}{n} \log \frac{n}{n_c},$$

where n_c is the number of occurrences of character c in T.

Definition 8 *Let $T_{1,n}$ be a text over an alphabet Σ. The* k-th order empirical entropy *of T is defined as*

$$H_k \;=\; H_k(T) \;=\; \sum_{s\in\Sigma^k, T^s\neq\epsilon} \frac{|T^s|}{n}\, H_0(T^s), \qquad (1)$$

where T^s is the subsequence of T formed by all the characters that occur followed by the context s in T. In order to have a context for the last k characters of T, we pad T with k characters "\$" (in addition to $T_n = \$$). More precisely, if the occurrences of s in $T_{2,n}\$^k$ start at positions $p_1, p_2, \ldots$, then $T^s = T_{p_1-1}T_{p_2-1}\ldots$.

In the text compression literature, it is customary to define T^s regarding the characters *preceded* by s, rather than *followed* by s. We use the reverse definition for technical convenience. Although the empirical entropy of T and its reverse do not necessarily match, the difference is relatively small [Ferragina and Manzini 2005], and if this is still an issue, one can always work on the reverse text.

The empirical entropy of a text T provides a lower bound to the number of bits needed to compress T using any compressor that encodes each character considering only the context of k characters that follow it in T. Many self-indexes state their space requirement as a function of the empirical entropy of the indexed text. This is useful because it gives a measure of the index size with respect to the size the

[3] Actually, the same formula of Manzini [2001] is used by Grossi et al. [2003], yet it is interpreted in this latter sense.

best k-th order compressor would achieve, thus relating the index size with the compressibility of the text.

We note that the classical entropy defined over infinite streams is always constant and can be zero. In contrast, the definition of H_k we use for finite texts is always positive, yet it can be $o(1)$ on compressible texts. For an extreme example, consider $T = abab \ldots ab\$$, where $H_0(T) = 1 - O(\log n/n)$ and $H_k(T) = 2/n$ for $k \geq 1$.

When we have a binary sequence $B[1, n]$ with κ bits set, it is good to remember some bounds on its zero-order entropy, such as $\log \binom{n}{\kappa} \leq nH_0(B) \leq \log \binom{n}{\kappa} + O(\log \kappa)$, $\kappa \log \frac{n}{\kappa} \leq nH_0(B) \leq \kappa \log \frac{n}{\kappa} + \kappa \log e$, and $\kappa \log \frac{n}{\kappa} \leq nH_0(B) \leq \kappa \log n$.

5.2 Self-Repetitions in Suffix Arrays

Consider again the suffix tree for our example text $T =$ `"alabar a la alabarda$"` depicted in Fig. 2 (page 8). Observe, for example, that the subtree rooted at `"abar"` contains leaves $\{3, 15\}$, while the subtree rooted at `"bar"` contains leaves $\{4, 16\}$, that is, the same positions shifted by one. The reason is simple: every occurrence of `"bar"` in T is also an occurrence of `"abar"`. Actually, the chain is longer: If one looks at subtrees rooted at `"alabar"`, `"labar"`, `"abar"`, `"bar"`, `"ar"`, and `"r"`, the same phenomenon occurs, and positions $\{1, 13\}$ become $\{6, 18\}$ after five steps. The same does not occur, for example, with the subtree rooted at `" a"`, whose leaves $\{7, 12\}$ do not repeat as $\{8, 13\}$ inside another subtree. The reason is that not all occurrences of `"a"` start within occurrences of `" a"` in T, and thus there are many more leaves rooted by `"a"` in the suffix tree, apart from 8 and 13.

Those repetitions show up in the suffix array A of T, depicted in Fig. 3. For example, consider $A[18, 19]$ with respect to $A[10, 11]$: $A[18] = 2 = A[10] + 1$ and $A[19] = 14 = A[11] + 1$. We denote such relationship by $A[18, 19] = A[10, 11] + 1$. There are also longer regularities that do not correspond to a single subtree of the suffix tree, for example $A[18, 21] = A[10, 13] + 1$. Still, the text property responsible for the regularity is the same: All the text suffixes in $A[10, 13]$ start with `"a"`, while those in $A[18, 21]$ are the same suffixes with the initial `"a"` excluded. The regularity appears because, for each pair of consecutive suffixes aX and aY in $A[10, 13]$, the suffixes X and Y are contiguous in $A[18, 21]$, that is, there is no other suffix W such that $X < W < Y$ elsewhere in the text. This motivates the definition of *self-repetition* initially devised by Mäkinen [2000, 2003].

Definition 9 *Given a suffix array A, a* self-repetition *is an interval $[i, i+\ell]$ of $[1, n]$ such that there exists another interval $[j, j+\ell]$ satisfying $A[j+r] = A[i+r]+1$ for all $0 \leq r \leq \ell$. For technical convenience, cell $A[1] = n$ is taken as a self-repetition of length 1, whose corresponding j is such that $A[j] = 1$.*

A measure of the amount of regularity in a suffix array is how many self-repetitions we need to cover the whole array. This is captured by the following definition [Mäkinen and Navarro 2004a, 2005a, 2005b].

Definition 10 *Given a suffix array A, we define n_{sr} as the minimum number of self-repetitions necessary to cover the whole A. This is the minimum number of nonoverlapping intervals $[i_s, i_s + \ell_s]$ that cover the interval $[1, n]$ such that, for any s, there exists j_s satisfying $A[j_s + r] = A[i_s + r] + 1$ for all $0 \leq r \leq \ell_s$ (except for $i_1 = 1$, where $\ell_1 = 0$ and $A[j_1] = 1$).*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A =	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18
	1	2	3	4	5		6		7				8		9		10		11	12	13
Ψ =	10	7	11	17	1	3	4	14	15	18	19	20	21	12	13	5	6	8	9	2	16
T^{bwt} =	a	r	a	a	d	l	_	l	l	$	_	b	b	a	a	r	_	a	a	a	a

Fig. 7. The suffix array A of the text T = "alabar a la alabarda$" and its corresponding function Ψ. Below A we show the minimal cover with with self-repetitions, and below Ψ we show the runs. Both coincide. On the bottom, the characters of T^{bwt}, where we show the equal-letter runs. Almost all *targets* of self-repetitions become equal-letter runs.

The suffix array A in Fig. 7 illustrates, where the covering is drawn below A. The 8th interval, for example, is $[i_8, i_8 + \ell_8] = [10, 13]$, corresponding to $j_8 = 18$.

Self-repetitions are best highlighted through the definition of function Ψ (recall Section 4.4), which tells where in the suffix array lies the pointer following the current one [Grossi and Vitter 2000].

Definition 11 *Given suffix array $A[1, n]$,* function $\Psi : [1, n] \rightarrow [1, n]$ *is defined so that, for all $1 \le i \le n$, $A[\Psi(i)] = A[i] + 1$. The exception is $A[1] = n$, in which case we require $A[\Psi(1)] = 1$ so that Ψ is actually a permutation.*

Function Ψ is heavily used in most compressed suffix arrays, as seen later. There are several properties of Ψ that make it appealing to compression. A first one establishes that Ψ is monotonically increasing in the areas of A that point to suffixes starting with the same character [Grossi and Vitter 2000].

Lemma 1 *Given a text $T_{1,n}$, its suffix array $A[1, n]$, and the corresponding function Ψ, it holds $\Psi(i) < \Psi(i + 1)$ whenever $T_{A[i]} = T_{A[i+1]}$.*

To see that the lemma holds, assume that $T_{A[i],n} = cX$ and $T_{A[i+1],n} = cY$, so $cX < cY$ and then $X < Y$. Thus $T_{A[i]+1,n} = T_{A[\Psi(i)],n} = X$ and $T_{A[i+1]+1,n} = T_{A[\Psi(i+1)],n} = Y$. So $T_{A[\Psi(i)],n} < T_{A[\Psi(i+1)],n}$, and thus $\Psi(i) < \Psi(i + 1)$.

Another interesting property is a special case of the above: how does Ψ behave inside a self-repetition $A[j+r] = A[i+r]+1$ for $0 \le r \le \ell$. Note that $\Psi(i+r) = j+r$ throughout the interval. The first two arrays in Fig. 7 illustrate. This motivates the definition of *runs in* Ψ [Mäkinen and Navarro 2004a, 2005a, 2005b].

Definition 12 *A* run in Ψ *is a maximal interval $[i, i + \ell]$ in sequence Ψ such that $\Psi(i + r + 1) - \Psi(i + r) = 1$ for all $0 \le r < \ell$.*

Note that the number of runs in Ψ is n minus the number of positions i such that $\Psi(i + 1) - \Psi(i) = 1$. The following lemma should not be surprising [Mäkinen and Navarro 2004a, 2005a, 2005b].

Lemma 2 *The number of self-repetitions n_{sr} to cover a suffix array A is equal to the number of runs in the corresponding Ψ function.*

5.3 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform [Burrows and Wheeler 1994] is a reversible transformation from strings to strings[4]. This transformed text is easier to compress by local optimization methods [Manzini 2001].

Definition 13 *Given a text $T_{1,n}$ and its suffix array $A[1,n]$, the* Burrows-Wheeler transform (BWT) *of T, $T^{bwt}_{1,n}$, is defined as $T^{bwt}_i = T_{A[i]-1}$, except when $A[i] = 1$, where $T^{bwt}_i = T_n$.*

That is, T^{bwt} is formed by sequentially traversing the suffix array A and concatenating the characters that *precede* each suffix. Fig. 7 illustrates.

A useful alternative view of the BWT is as follows. A *cyclic shift* of $T_{1,n}$ is any string of the form $T_{i,n}T_{1,i-1}$. Let M be a matrix containing all the cyclic shifts of T in lexicographical order. Let F be the first and L the last column of M. Since T is terminated with $T_n = \$$, which is smaller than any other, the cyclic shifts $T_{i,n}T_{1,i-1}$ are sorted exactly like the suffixes $T_{i,n}$. Thus M is essentially the suffix array A of T, F is a sorted list of all the characters in T, and L is the list of characters preceding each suffix, that is, $L = T^{bwt}$.

Fig. 8 illustrates. Note that row $M[i]$ is essentially $T_{A[i],n}$ of Fig. 4, and that every column of M is a permutation of the text. Among those, L can be reversed back to the text, and in addition it exhibits some compressibility properties that can be exploited in many ways, as we show next.

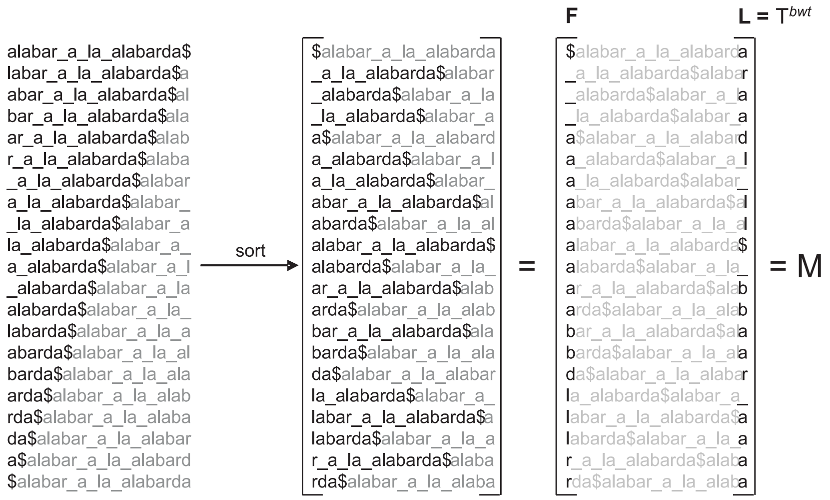

Fig. 8. Obtaining the BWT for the text `"alabar a la alabarda$"`.

In order to reverse the BWT, we need to be able to know, given a character in L, where it appears in F. This is called the *LF-mapping* [Burrows and Wheeler 1994; Ferragina and Manzini 2000] (recall Section 4.1).

[4] Actually the original string must have a unique endmarker for the transformation to be reversible. Otherwise one must know the position of the original last character in the transformed string.

Definition 14 *Given strings F and L resulting from the* BWT *of text T, the* LF-mapping *is a function $LF : [1,n] \longrightarrow [1,n]$, such that $LF(i)$ is the position in F where character L_i occurs.*

Consider the single occurrence of character "d" in Fig. 8. It is at L_5. It is easy to see where it is in F: Since F is alphabetically sorted and there are 15 characters smaller than "d" in T, it must be F_{16} = "d", thus $LF(5) = 16$. The situation is a bit more complicated for L_{18} = "a", because there are several occurrences of the same character. Note, however, that all the occurrences of "a" in F are sorted according to the suffix that follows the "a". Likewise, all the occurrences of "a" in L are also sorted accordingly to the suffix that follows that "a". Therefore, equal characters preserve the same order in F and L. As there are 4 characters smaller than "a" in T and 6 occurrences of "a" in $L_{1,18}$, we have that L_{18} = "a" occurs at $F_{4+6} = F_{10}$, that is, $LF(18) = 10$.

The following lemma gives the formula for the LF-mapping [Burrows and Wheeler 1994; Ferragina and Manzini 2000].

Lemma 3 *Let $T_{1,n}$ be a text and F and L be the result of its* BWT*. Let $C : \Sigma \longrightarrow [1,n]$ and $Occ : \Sigma \times [1,n] \longrightarrow [1,n]$, such that $C(c)$ is the number of occurrences in T of characters alphabetically smaller than c, and $Occ(c,i)$ is the number of occurrences of character c in $L_{1,i}$. Then, it holds $LF(i) = C(L_i) + Occ(L_i, i)$.*

With this mapping, reversing the BWT is simple, as L_i always precedes F_i in T. Since $T_n = \$$, the first cyclic shift in M is $T_n T_{1,n-1}$, and therefore $T_{n-1} = L_1$. We now compute $i = LF(1)$ to learn that T_{n-1} is at F_i, and thus $T_{n-2} = L_i$ precedes $F_i = T_{n-1}$. With $i' = LF(i)$ we learn that T_{n-2} is at $F_{i'}$ and thus $T_{n-3} = L_{i'}$, and so on, $T_{n-k} = L[LF^{k-1}(1)]$.

We finish with an observation that is crucial to understand the relation between different kinds of existing self-indexes [Sadakane 2000; Ferragina and Manzini 2000].

Lemma 4 *Functions LF and Ψ are the inverse of each other.*

To see this, note that $LF(i)$ is the position in F of character $L_i = T_i^{bwt} = T_{A[i]-1}$, or which is the same, the position in A that points to suffix $T_{A[i]-1,n}$. Thus $A[LF(i)] = A[i]-1$, or $LF(i) = A^{-1}[A[i]-1]$. On the other hand, according to Definition 11, $A[\Psi(i)] = A[i]+1$. Hence $LF(\Psi(i)) = A^{-1}[A[\Psi(i)]-1] = A^{-1}[A[i]+1-1] = i$ and vice versa. The special case for $A[1] = n$ works too.

5.4 Relation between Regularities and Compressibility

We start by pointing out a simple but essential relation between T^{bwt}, the Burrows-Wheeler transform of T, and $H_k(T)$, the k-th order empirical entropy of T. Note that, for each text context s of length k, all the suffixes starting with that context appear consecutively in A. Therefore, the characters that precede each context (which form T^s) appear consecutively in T^{bwt}. The following lemma [Ferragina et al. 2005; Ferragina et al. 2004] shows that it suffices to compress the characters of each context to their zero-order entropy to achieve k-th order entropy overall.

Theorem 4 *Given $T_{1,n}$ over an alphabet of size σ, if we divide T^{bwt} into (at most) σ^k pieces according to the text context that follows each character in T, and then* *compress each piece T^s corresponding to context s using* $c|T^s|H_0(T^s)+f(|T^s|)$ *bits, where the f is a concave function*[5]*, then the representation for the whole T^{bwt} requires at most $cnH_k(T)+\sigma^k f(n/\sigma^k)$ bits.*

To see that the theorem holds, it is enough to recall Eq. (1), as we are representing the characters T^s followed by each of the contexts $s \in \Sigma^k$ using space proportional to $|T^s|H_0(T^s)$. The extra space, $\sigma^k f(n/\sigma^k)$, is just the worst case of the sum of σ^k values $f(|T^s|)$ where the values $|T^s|$ add up to n.

Thus, T^{bwt} is the concatenation of all the T^s. It is enough to encode each such portion of T^{bwt} with a zero-order compressor to obtain a k-th order compressor for T, for any k. The price of using a longer context (larger k) is paid in the extra $\sigma^k f(n/\sigma^k)$ term. This can be thought of as the price to manage the model information, and it can easily dominate the overall space if k is not small enough.

Let us now consider the number of equal-letter runs in T^{bwt}. This will be related both to self-repetitions in suffix arrays and to the empirical entropy of T [Mäkinen and Navarro 2004a, 2005a, 2005b].

Definition 15 *Given T^{bwt}, the* BWT *of a text T, n_{bw} is the number of* equal-letter runs *in T^{bwt}, that is, n minus the number of positions j such that $T^{bwt}_{j+1} = T^{bwt}_j$.*

There is a close tie between the runs in Ψ (or self-repetitions in A), and the equal-letter runs in T^{bwt} [Mäkinen and Navarro 2004a, 2005a, 2005b].

Lemma 5 *Let n_{sr} be the number of runs in Ψ (or self-repetitions in A), and let n_{bw} be the number of equal-letter runs in T^{bwt}, all with respect to a text T over an alphabet of size σ. Then it holds $n_{sr} \le n_{bw} \le n_{sr}+\sigma$.*

To see why the lemma holds, consider Figs. 3 (page 9) and 7. Let us focus on the longest self-repetition, $A[10,13] = \{1,13,5,17\}$. All those suffixes (among others) start with "a". The self-repetition occurs in $\Psi([10,13]) = [18,21]$, that is, $A[18,21] = \{2,14,6,18\}$. All those suffixes are preceded by "a", because all $\{1,13,5,17\}$ start with "a". Hence there is a run of characters "a" in $T^{bwt}_{18,21}$.

It should be obvious that in all cases where all the suffixes of a self-repetition start with the same character, there must be an equal-letter run in T^{bwt}: Let $A[j+r] = A[i+r]+1$ and $T_{A[i+r]} = c$ for $0 \le r \le \ell$. Then $T^{bwt}_{j+r} = T_{A[j+r]-1} = T_{A[i+r]} = c$ holds for $0 \le r \le \ell$. On the other hand, because of the lexicographical ordering, consecutive suffixes change their first character at most σ times throughout $A[1,n]$. Thus, save at most σ exceptions, every time $\Psi(i+1)-\Psi(i) = 1$ (that is, we are within a self-repetition), there will be a distinct $j = \Psi(i)$ such that $T^{bwt}_{j+1} = T^{bwt}_j$. Thus $n_{bw} \le n_{sr}+\sigma$. (There is one such exception in Fig. 7, where the self-repetition $A[16,17]+1 = A[5,6]$ does not correspond to an equal-letter run in $T^{bwt}_{5,6}$.)

On the other hand, every time $T^{bwt}_{j+1} = T^{bwt}_j$, we know that suffix $T_{A[j],n} = X$ is followed by $T_{A[j+1],n} = Y$, and both are preceded by the same character c. Hence suffixes cX and cY must also be contiguous, at positions i and $i+1$ so that $\Psi(i) = j$ and $\Psi(i+1) = j+1$, thus it holds $\Psi(i+1)-\Psi(i) = 1$ for a distinct i every time $T^{bwt}_{j+1} = T^{bwt}_j$. Therefore, $n_{sr} \le n_{bw}$. These observations prove Lemma 5.

[5]That is, its second derivative is never positive.

We finally relate the k-th order empirical entropy of T with n_{bw} [Mäkinen and Navarro 2004a, 2005a, 2005b].

Theorem 5 *Given a text $T_{1,n}$ over an alphabet of size σ, and given its* BWT *T^{bwt}, with n_{bw} equal-letter runs, it holds $n_{bw} \leq nH_k(T)+\sigma^k$ for any $k \geq 0$. In particular, it holds $n_{bw} \leq nH_k(T)+o(n)$ for any $k \leq \log_\sigma n - \omega(1)$. The bounds are obviously valid for $n_{sr} \leq n_{bw}$ as well.*

We only attempt to give a flavor of why the theorem holds. The idea is to partition T^{bwt} according to the contexts of length k. Following Eq. (1), $nH_k(T)$ is the sum of zero-order entropies over all the T^s strings. It can then be shown that, within a single $T^{bwt}_{i,j} = T^s$, the number of equal-letter runs in T^s can be upper bounded in terms of the zero-order entropy of the string T^s. A constant $f(|S|) = 1$ in the upper bound is responsible for the σ^k overhead, which is the number of possible contexts of length k. Thus the rest is a consequence of Theorem 4.

6. BASIC COMPACT DATA STRUCTURES

We will learn later that nearly all approaches to represent suffix arrays in compressed form take advantage of *compressed representations of binary sequences*. That is, we are given a bit vector (or bit string) $B_{1,n}$, and we want to compress it while still supporting several operations on it. Typical operations are:

B_i:	Accesses the i-th element.
$rank_b(B,i)$:	Returns the number of times bit b appears in the prefix $B_{1,i}$.
$select_b(B,j)$:	Returns the position i of the j-th appearance of bit b in $B_{1,n}$.

Other useful operations are $prev_b(B,i)$ and $next_b(B,i)$, which give the position of the previous/next bit b from position i. However, these operations can be expressed via *rank* and *select*, and hence are usually not considered separately. Notice also that $rank_0(B,i) = i - rank_1(B,i)$, so considering $rank_1(B,i)$ is enough. However, the same duality does not hold for *select*, and we have to consider both $select_0(B,j)$ and $select_1(B,j)$. We call a representation of B *complete* if it supports all the listed operations in constant time, and *partial* if it supports them only for 1-bits, that is, if it supports $rank_b(B,i)$ only if $B_i = 1$ and it only supports $select_1(B,j)$.

The study of succinct representations of various structures, including bit vectors, was initiated by Jacobson [1989]. The main motivation to study these operations came from the possibility to simulate tree traversals in small space: It is possible to represent the shape of a tree as a bit vector, and then the traversal from a node to a child and vice versa can be expressed via constant number of *rank* and *select* operations. Jacobson showed that attaching a dictionary of size $o(n)$ to the bit vector $B_{1,n}$ is sufficient to support *rank* operation in constant time on the RAM model. He also studied *select* operation, but for the RAM model the solution was not yet optimal. Later, Munro [1996] and Clark [1996] obtained constant-time complexity for *select* on the RAM model, using also $o(n)$ extra space.

Although $n + o(n)$ bits are asymptotically optimal for incompressible binary sequences, one can obtain more space-efficient representations for compressible ones. Consider, for example, $select_1(B,i)$ operation on a bit vector containing $\kappa = o(n/\log n)$ 1-bits. One can directly store all answers in $O(\kappa \log n) = o(n)$ bits.

Pagh [1999] was the first to study compressed representations of bit vectors supporting more than just access to B_i. He gave a representation of bit vector $B_{1,n}$ that uses $\lceil \log \binom{n}{\kappa} \rceil + o(\kappa) + O(\log\log n)$ bits. In principle this representation supported only B_i queries, yet it also supported *rank* queries for sufficiently dense bit vectors, $n = O(\kappa \,\mathrm{polylog}(\kappa))$. Recall that $\log \binom{n}{\kappa} = nH_0(B) + O(\log n)$.

This result was later enhanced by Raman, Raman, and Rao [2002], who developed a representation with similar space complexity, $nH_0(B) + o(\kappa) + O(\log\log n)$ bits, supporting *rank* and *select*. However, this representation is *partial*. They also provide a new *complete* representation requiring $nH_0(B)+O(n\log\log n/\log n)$ bits.

Recent lower bounds [Golynski 2006] show that these results can hardly be improved, as $\Omega(n \log\log n/\log n)$ is a lower bound on the extra space of any rank/select index achieving $O(\log n)$ time if the index stores B explicitly. For compressed representations of B, one needs $\Omega((\kappa/\tau)\log\tau)$ bits of space (overall) to answer queries in time $O(\tau)$. This latter bound still leaves some space for improvements.

In the rest of this section we explain the most intuitive of these results, to give a flavor of how some of the solutions work. We also show how the results are extended to non-binary sequences and two-dimensional searching. Most implementations of these solutions sacrifice some theoretical guarantees but work well in practice [Geary et al. 2004; Kim et al. 2005; González et al. 2005; Sadakane and Okanohara 2006].

6.1 Basic $n + o(n)$-bit Solutions for Binary Sequences

We start by explaining the $n + o(n)$ bits solution supporting $rank_1(B,i)$ and $select_1(B,j)$ in constant time [Jacobson 1989; Munro 1996; Clark 1996]. Then we also have $rank_0(B,i) = i - rank_1(B,i)$, and $select_0(B,j)$ is symmetric.

Let us start with *rank*. The structure is composed of a two-level dictionary with partial solutions (directly storing the answers at regularly sampled positions i), plus a global table storing answers for every possible short binary sequence. The answer to a *rank* query is formed by summing values from these dictionaries and tables.

For clarity of presentation we assume n is a power of four. The general case is handled by considering floors and ceilings when necessary. We assume all divisions x/y to give the integer $\lfloor x/y \rfloor$.

Let us start from the last level. Consider a substring `smallblock` of $B_{1,n}$ of length $t = \frac{\log n}{2}$. This case is handled by the so-called four-Russians technique [Arlazarov et al. 1975]: We build a table $\texttt{smallrank}[0,\sqrt{n}-1][0,t-1]$ storing all answers to *rank* queries for all binary sequences of length t (note that $2^t = \sqrt{n}$). Then $rank_1(\texttt{smallblock}, i) = \texttt{smallrank}[\texttt{smallblock}, i]$ is obtained in constant time. To index `smallrank`, `smallblock` is regarded as an integer in the usual way. Note that this can be extended to substrings of length $c\log n$, which would be solved in at most $2c$ accesses to table `smallrank`. For example, if `smallblock` is of length $\log n$, then $rank_1(\texttt{smallblock}, t+3) = \texttt{smallrank}[\texttt{half}^1, t] + \texttt{smallrank}[\texttt{half}^2, 3]$, where $\texttt{half}^1$ and $\texttt{half}^2$ are the two halves of `smallblock`.

We could complete the solution by dividing B into *blocks* of length, say, $2t$, and explicitly storing *rank* answers for block boundaries, in a table $\texttt{sampledrank}[0, \frac{n}{\log n} - 1]$, such that $\texttt{sampledrank}[q] = rank_1(B, q\log n)$ for $0 \le q < \frac{n}{\log n}$. Then, given $i = q\log n + r$, $0 \le r < \log n$, we can express $rank_1(B,i) = \texttt{sampledrank}[q] + rank_1(B[q\log n + 1, q\log n + \log n], r)$. As the latter $rank_1$ query is answered in constant time using table `smallrank`, we have constant-time *rank* queries on B.

The problem with `sampledrank` is that there are $\frac{n}{\log n}$ blocks in B, each of which requires $\log n$ bits in `sampledrank`, for n total bits. To obtain $o(n)$ extra space, we build a *superblock* dictionary $\texttt{superblockrank}[0, \frac{n}{\log^2 n} - 1]$ such that $\texttt{superblockrank}[q'] = rank_1(B, q' \log^2 n)$ for $0 \le q' < \frac{n}{\log^2 n}$. We replace structure `sampledrank` with `blockrank`, which stores *relative* answers inside its superblock. That is, $\texttt{blockrank}[q] = \texttt{sampledrank}[q] - \texttt{superblockrank}[\frac{q}{\log n}]$ for $0 \le q < \frac{n}{\log n}$. Then, for $i = q' \log^2 n + r' = q \log n + r$, $0 \le r' < \log^2 n$, $0 \le r < \log n$, $rank_1(B, i) = \texttt{superblockrank}[q'] + \texttt{blockrank}[q] + rank_1(B[q \log n + 1, q \log n + \log n], r)$, where the last $rank_1$ query is answered in constant time using table `smallrank`.

The values stored in `blockrank` are in the range $[0, \log^2 n - 1]$, hence table `blockrank` takes $O(n \log\log n / \log n)$ bits. Table `superblockrank` takes $O(n/\log n)$ bits, and finally table `smallrank` takes $O(\sqrt{n} \log n \log\log n)$ bits. We have obtained the claimed $n + o(n)$ bits representation of $B_{1,n}$ supporting constant time *rank*. Fig. 9 illustrates the structure.

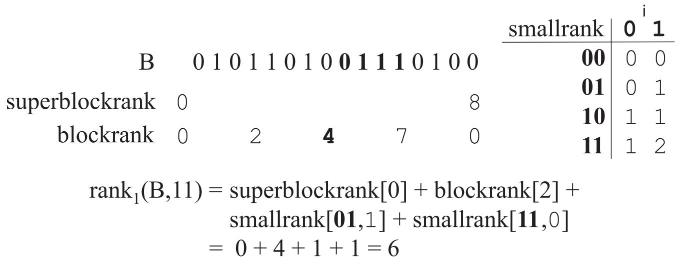

Fig. 9. An example of constant-time *rank* computation using $n + o(n)$ bits of space.

Extending the structure to provide constant-time *select* is more complicated. We explain here a version simplified from [Munro 1996; Clark 1996].

We partition the space $[1, n]$ of possible *arguments* of *select* (that is, values j of $select_1(B, j)$) into blocks of $\log^2 n$ arguments. A dictionary $\texttt{superblockselect}[j]$, requiring $O(n/\log n)$ bits, answers $select_1(B, j \log^2 n)$ in constant time.

Some of those blocks may span a large extent in B (with many 0-bits and just $\log^2 n$ 1-bits). A fundamental problem for using blocks and superblocks for *select* is that there is no guarantee that relative answers inside blocks do not require $\log n$ bits anyway. A block is called *long* if it spans more than $\log^4 n$ positions in B, and *short* otherwise. Note that there cannot be more than $n/\log^4 n$ long blocks. As long blocks are problematic, we simply store all their $\log^2 n$ answers explicitly. As each answer requires $\log n$ bits, this accounts for other $n/\log n$ bits overall.

The short blocks contain $\kappa = \log^2 n$ 1-bits (arguments for $select_1$) and span at most $\log^4 n$ bits in B. We divide them again into *miniblocks* of $\log^2 \kappa = O((\log\log n)^2)$ arguments. A miniblock directory $\texttt{miniblockselect}[j]$ will store the relative answer to $select_1$ inside the short block, that is, $\texttt{miniblockselect}[j] = select_1(B, j \log^2 \kappa) - \texttt{superblockselect}[\frac{j \log^2 \kappa}{\log^2 n}]$. Values in `miniblockselect` are in the range $[1, \log^4 n]$ and thus require $O(\log\log n)$ bits. Thus `miniblockselect` requires $O(n/\log\log n)$ bits. A miniblock will be called *long* if it spans more than $\log n$ bits in B. For long miniblocks, we will again store all their answers explicitly. There are at most $n/\log n$ long miniblocks overall, so storing all the $\log^2\kappa$ answers of all long miniblocks requires $O(n(\log\log n)^3/\log n)$ bits. Finally, short miniblocks span at most $\log n$ bits in B, so a precomputed table analogous to `smallrank` gives their answers using $O(\sqrt{n}\log n\log\log n)$ bits. This completes the solution, which has to be duplicated for $select_0(B,j)$.

Theorem 6 *Bit vector $B_{1,n}$ can be represented using $n+o(n)$ bits of space so that B_i, $rank_b(B,i)$, and $select_b(B,j)$, can be answered in constant time.*

6.2 More Sophisticated nH_0-bit Solutions

We explain now how to improve the representation of the previous section for compressible sequences, so as to obtain complete representations requiring $nH_0(B)+o(n)$ bits of space [Raman et al. 2002].

We cut B into blocks of fixed length $t=\frac{\log n}{2}$. Each such block $I=B_{ti+1,ti+t}$ with κ bits set will belong to class κ of t-bitmaps. For example, if $t=4$, then class 0 is $\{0000\}$, class 1 is $\{0001,0010,0100,1000\}$, class 2 is $\{0011,0101,0110,1001,1010,1100\}$, and so on until class $t=4$, $\{1111\}$. As class κ contains $\binom{t}{\kappa}$ elements, an index to identify a t-bitmap within its class requires only $\log\binom{t}{\kappa}$ bits. Instead of using t bits to represent a block, we use the pair (κ,r): the class identifier $0\le\kappa\le t$, using $\lceil\log(t+1)\rceil$ bits, and the index r within class κ, using $\lceil\log\binom{t}{\kappa}\rceil$ bits. Then B is represented as a sequence of $\lceil n/t\rceil$ such pairs.

The class identifiers amount to $O(n\log(t)/t)=O(n\log\log n/\log n)$ bits overall. The interesting part is the sequence of indexes. Let $\texttt{smallblock}^i$, be the i-th block, with κ_i bits set. The number of bits required by all the blocks is [Pagh 1999]

$$\left\lceil\log\binom{t}{\kappa_1}\right\rceil+\ldots+\left\lceil\log\binom{t}{\kappa_{\lceil n/t\rceil}}\right\rceil \quad\le\quad \log\left(\binom{t}{\kappa_1}\times\ldots\times\binom{t}{\kappa_{\lceil n/t\rceil}}\right)+n/t$$

$$\le\quad\log\binom{n}{\kappa_1+\ldots+\kappa_{\lceil n/t\rceil}}+n/t \quad=\quad \log\binom{n}{\kappa}+n/t \quad\le\quad nH_0(B)+O(n/\log n),$$

where the second inequality holds because the ways to choose κ_i bits from each block of t bits are included in the ways to choose $\kappa=\kappa_1+\ldots+\kappa_{\lceil n/t\rceil}$ bits out of n. Thus, we have represented B with $nH_0(B)+o(n)$ bits.

We need more structures to answer queries on B. The same `superblockrank` and `blockrank` directories used in Section 6.1, with block size t, are used. As the descriptions (κ,r) have varying length, we need position directories `superblockpos` and `blockpos`, which work like `superblockrank` and `blockrank` to give the position in the compressed B where the description of each block starts.

In order to complete the solution with table `smallrank`, we must index this table using the representations (κ,r), as bitmap `smallblock` is not directly available. For each class κ we store a table $\texttt{smallrank}_\kappa[r,i]$, giving $rank_1(\texttt{smallrank},i)$ for the block `smallrank` identified by pair (κ,r). In our example, $\texttt{smallrank}_2[4,2]=1$ as pair $(2,4)$ identifies bitmap 1001 and $rank(1001,2)=1$. Those $\texttt{smallrank}_\kappa$ tables need together $O(\sqrt{n}\log n\log\log n)$ bits, just as the original `smallrank`. Thus all the extra structures still require $o(n)$ bits. Fig. 10 illustrates.

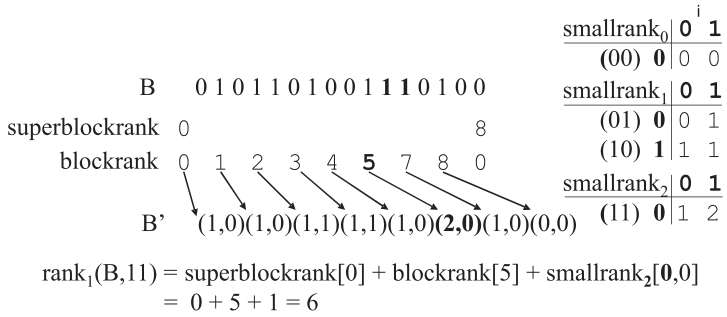

Fig. 10. Constant-time *rank* computation using $nH_0(B) + o(n)$ bits. B' is the sequence of pairs (κ, r), each occupying a variable number of bits. The values in `superblockpos` and `blockpos` (represented here by the arrows) point to the corresponding positions in the binary representation of B'. The arguments to `smallrank` show in parentheses the corresponding bit sequences.

For *select*, the solution is again more complicated, but it is essentially as in the previous section.

Theorem 7 *Bit vector $B_{1,n}$ can be represented using $nH_0(B)+O(n \log\log n/\log n)$ bits of space so that B_i, $rank_b(B,i)$, and $select_b(B,j)$, can be answered in constant time.*

6.3 Handling General Sequences and Wavelet Trees

Consider now a sequence (or string) $S_{1,n}$ from an alphabet of size $\sigma \geq 2$. We wish to support $rank_c(S,i)$ and $select_c(S,j)$ for all alphabet symbols c: $rank_c(S,i)$ gives the number of times character c appears in $S_{1,i}$ and $select_c(S,j)$ gives the position of the j-th c in S. Analogously to the binary case, we call a representation of S *complete* if it supports queries S_i, $rank_c(S,i)$, and $select_c(S,j)$, in constant time for all symbols c.

We can easily obtain a complete representation for S using the results from the previous section: Consider indicator bit vectors $B^c_{1,n}$ such that $B^c_i = 1$ iff $S_i = c$. Then $rank_c(S,i) = rank_1(B^c,i)$ and $select_c(S,j) = select_1(B^c,j)$. Using Theorem 7, the representations of vectors B^c take overall $\sum_{c\in\Sigma}(nH_0(B^c)+o(n)) = \sum_{c\in\Sigma}(n_c \log \frac{n}{n_c} + O(n_c) + o(n)) = nH_0(S) + O(n) + o(\sigma n)$ bits (end of Section 5.1).

The $O(n)$ extra term can be removed with a more careful design [Ferragina et al. 2006]. Essentially, one can follow the development leading to Theorem 7 on a general sequence. Now binomials become multinomials and the scheme is somewhat more complicated, but the main idea does not change.

Lemma 6 *Sequence $S_{1,n}$ over an alphabet of size σ can be represented using $nH_0(S)$ $+O(\sigma n \log\log n/\log_\sigma n)$ bits of space so that B_i, $rank_c(S,i)$, and $select_c(S,j)$, can be answered in constant time. Note that the extra space on top of $nH_0(S)$ is $o(n \log \sigma)$ only if the alphabet is very small, namely $\sigma = o(\log n/\log\log n)$.*

A completely different technique is the *wavelet tree* [Grossi et al. 2003]. Fig. 5, on page 13, illustrates this structure. The wavelet tree is a perfect binary tree of height $\lceil \log \sigma \rceil$, built on the alphabet symbols, such that the root represents the whole alphabet and each leaf represents a distinct alphabet symbol. If a node v represents alphabet symbols in the range $\Sigma^v = [i, j]$, then its left child v_l represents $\Sigma^{v_l} = [i, \frac{i+j}{2}]$ and its right child v_r represents $\Sigma^{v_r} = [\frac{i+j}{2} + 1, j]$.

We associate to each node v the subsequence S^v of S formed by the symbols in Σ^v. Yet, S^v is not really stored at the node. We just store a bit sequence B^v telling whether symbols in S^v go left or right: $B^v_i = 1$ iff $S^v_i \in \Sigma^{v_r}$ (i.e., S^v_i goes right).

All queries on S are easily answered in $O(\log \sigma)$ time with the wavelet tree, provided we have complete representations of the bit vectors B^v. To determine S_i, we check B^{root}_i to decide whether to go left or right. If we go left, we now seek the character at position $rank_0(B^{root}, i)$ in the left child of the root, otherwise we seek the character at position $rank_1(B^{root}, i)$ in its right child. We continue recursively until reaching a leaf corresponding to a single character, which is the original S_i.

Similarly, to answer $rank_c(S, i)$, we go left or right, adapting i accordingly. This time we choose left or right depending on whether character c belongs to Σ^{v_l} or Σ^{v_r}. Once we arrive at a leaf, the current i value is the answer. Fig. 5 gives an example for $rank_{\mathtt{a}}(S, 16) = 5$.

Finally, to answer $select_c(S, j)$, we start at the leaf corresponding to c and move upwards in the tree. If the leaf is a left child, then the position corresponding to j in its parent v is $select_0(B^v, j)$, otherwise it is $select_1(B^v, j)$. When we reach the root, the current j value is the answer. For example, in Fig. 5, $select_{\mathtt{a}}(S, 5)$ starts with the leaf for "a". It is a left child, so in its parent the position is $select_0(00011000000, 5) = 7$. This in turn is a right child, so in its parent the position is $select_1(111000111101111, 7) = 10$. We finish with answer 15 at the root.

If we use the complete representation of Theorem 6 for the bit vectors B^v, the overall space is $n \log \sigma (1 + o(1))$, that is, essentially the same space to store S (and we do not need to also store S). Yet, by using the representation of Theorem 7, the sum of entropies of all bit vectors simplifies to $nH_0(S)$ and the extra terms add up $O(n \log\log n / \log_\sigma n) = o(n \log \sigma)$ [Grossi et al. 2003].

Theorem 8 *Sequence $S_{1,n}$ over an alphabet of size σ can be represented using the wavelet tree in $nH_0(S) + O(n \log\log n / \log_\sigma n)$ bits of space, so that S_i, $rank_c(S, i)$, and $select_c(S, j)$, can be answered in $O(\log \sigma)$ time.*

It is possible to combine the representations of Lemma 6 and Theorem 8. The former gives a complete representation (constant query time) with sublinear extra space, for $\sigma = o(\log n / \log\log n)$. The latter works for any alphabet σ but it pays $O(\log \sigma)$ query time. The idea is to use r-ary wavelet trees. Instead of storing bitmaps at each node, we now store sequences over an alphabet of size r to represent the tree branch chosen by each character. Those sequences are handled with Lemma 6 [Ferragina et al. 2006]. By carefully choosing r, one gets constant access time for $\sigma = O(\mathrm{polylog}(n))$, and improved access time for larger alphabets.

Theorem 9 *Sequence $S_{1,n}$ over an alphabet of size $\sigma = o(n)$, can be represented using a multi-ary wavelet tree in $nH_0(S) + o(n \log \sigma)$ bits of space, so that S_i, $rank_c(S, i)$, and $select_c(S, j)$, are answered in $O(1 + \log \sigma / \log\log n)$ time.*

Finally, some very recent work [Golynski et al. 2006] obtains $O(\log\log \sigma)$ time for queries S_i and $rank$, and constant time for $select$, using $n \log \sigma (1 + o(1))$ bits.

6.4 Two-dimensional Range Searching

As we will see later, some compressed indexes reduce some search subproblems to two-dimensional range searching. We present here one classical data structure by Chazelle [Chazelle 1988]. For simplicity we focus on the problem variant that arises in our application: One has a set of n points over an $n \times n$ grid, such that there is exactly one point for each row i and one for each column j.

Let us regard the set of n points as a sequence $S = i(1)i(2)\ldots i(n)$, so that $i(j)$ is the row of the only point at column j. As all the rows are also different, S is actually a permutation of the interval $[1, n]$. More complex scenarios can be reduced to this simplified setting.

The most succinct way of describing Chazelle's data structure is to say that it is the wavelet tree of S, so the tree partitions the point set into two halves according to their row value i. Thus the structure can be represented using $n \log n(1 + o(1))$ bits of space. Yet, the query algorithms are rather different.

Let us focus on retrieving all the points that lie within a two-dimensional range $[i, i'] \times [j, j']$. Let $B_{1,n}$ be the bitmap at the tree root. We can *project* the range $[j, j']$ onto the left child as $[j_l, j'_l] = [rank_0(B, j-1)+1, rank_0(B, j')]$, and similarly onto the right child with $rank_1$. We backtrack over the tree, abandoning a path at node v either when the local interval $[j_v, j'_v]$ is empty, or when the local interval $[i_v, i'_v]$ does not intersect anymore the original $[i, i']$. If we arrive at a leaf $[i, i]$ without discarding it, then the point with row value i is part of the answer. In the worst case every answer is reported in $O(\log n)$ time, and we need $O(\log n)$ time if we want just to count the number of occurrences.

There exist other data structures [Alstrup et al. 2000] that require $O(n \log^{1+\gamma} n)$ bits of space, for any constant $\gamma > 0$, and can, after spending $O(\log \log n)$ time for the query, retrieve each occurrence in constant time. Another structure in the same paper takes $O(n \log n \log \log n)$ bits of space and requires $O((\log \log n)^2)$ time for the query, after which it can retrieve each answer in $O(\log \log n)$ time.

7. COMPRESSED SUFFIX ARRAYS

The first type of compressed indexes we are going to review can be considered as the result of *abstract optimization* of the suffix array data structure. That is, the search algorithm remains essentially as in Alg. 1, but suffix array A is taken as an abstract data type that gives access to the array in some way. This abstract data type is implemented using as little space as possible. This is the case of the *Compact Suffix Array* of Mäkinen [2000, 2003] (Mak-CSA) and the *Compressed Suffix Array* of Grossi and Vitter [2000, 2006] (GV-CSA). Both ideas appeared simultaneously and independently during the year 2000, and they are based on different ways of exploiting the regularities that appear on the suffix arrays of compressible texts. Those structures are still not self-indexes, as they need the text T to operate.

The Mak-CSA is mainly interesting by its property of using only *structural compression*, where one seeks the minimum size representation under a given family. For example, minimization of automata is structural compression. The Mak-CSA is basically a minimized array, where self-repetitions $A[i, i+\ell]$ are replaced by a link to the corresponding area $A[j, j+\ell]$ (recall Definition 9). From Theorem 5 it is easy to see that the size of the index is $O(nH_k(T) \log n)$ bits, which makes it the first compressed full-text index that existed. We will not detail it here, as the forthcoming results build on non-structural compression and supersede this one.

7.1 GV-CSA: Grossi and Vitter's Compressed Suffix Array

The *Compressed Suffix Array* of Grossi and Vitter [2000, 2006] (GV-CSA) is a succinct index based on the idea of providing access to $A[i]$ without storing A, so that the search algorithm is exactly as in Alg. 1. The text T is maintained explicitly.

The GV-CSA uses a hierarchical decomposition of A based on function Ψ (Definition 11). Let us focus on the first level of that hierarchical decomposition. Let $A_0 = A$ be the original suffix array. A bit vector $B^0[1,n]$ is defined so that $B^0[i] = 1$ iff $A[i]$ is even. Let also $\Psi_0[1, \lceil n/2 \rceil]$ contain the sequence of values $\Psi(i)$ for arguments i where $B^0[i] = 0$. Finally, let $A_1[1, \lfloor n/2 \rfloor]$ be the subsequence of $A_0[1,n]$ formed by the even $A_0[i]$ values, divided by 2.

Then, $A = A_0$ can be represented using only Ψ_0, B^0, and A_1. To retrieve $A[i]$, we first see if $B^0[i] = 1$. If it is, then $A[i]$ is (divided by 2) somewhere in A_1. The exact position depends on how many 1's are there in B^0 up to position i, that is, $A[i] = 2 \cdot A_1[rank_1(B^0, i)]$. If $B^0[i] = 0$, then $A[i]$ is odd and not represented in A_1. However, $A[i] + 1 = A[\Psi(i)]$ has to be even and thus represented in A_1. Since Ψ_0 collects only the Ψ values where $B^0[i] = 0$, we have $A[\Psi(i)] = A[\Psi_0[rank_0(B^0, i)]]$. Once we compute $A[\Psi(i)]$ (for even $\Psi(i)$), we simply obtain $A[i] = A[\Psi(i)] - 1$.

Fig. 11 illustrates. For example, to obtain $A[11]$, we verify that $B^0[11] = 0$, thus it is not represented in A_1. So we need to obtain $\Psi(11)$, which is $\Psi_0[rank_0(B^0, 11)] = \Psi_0[8] = 19$. We must then obtain $A[19]$. Not surprisingly, $B^0[19] = 1$, so $A[19]$ is at $A_1[rank_1(B^0, 19)] = A_1[8] = 7$. Thus $A[19] = 2 \cdot 7 = 14$ and finally $A[11] = A[19] - 1 = 13$. (Note the exception that $A[1]$ is odd and $A[\Psi(1)] = A[10]$ is odd too, but this is not a problem because we know that $A[1] = n$ always.)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A_0 =	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18
B^0 =	0	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	1	1	1	1	1
	1	2		3		4		5	6	7	8	9	10			11					
Ψ_0 =	10	7		17		3		14	15	18	19	20	21			5					
			1		2		3							4	5		6	7	8	9	10
A_1 =			6		10		4							2	8		5	1	7	3	9

Fig. 11. The first level of the GV-CSA recursive structure, for the text `"alabar a la alabarda$"`. We show the original suffix array $A = A_0$ and structures B^0, Ψ_0, and A_1. We show Ψ_0 and A_1 in scattered form to ease understanding but they are obviously stored contiguously. Recall that A_0 is not really stored but replaced by the other three structures.

The idea can be used recursively: Instead of representing A_1, we replace it with B^2, Ψ_2, and A_2. This is continued until A_h is small enough to be represented explicitly. Alg. 2 gives the pseudocode to extract an entry $A[i]$ from the recursive structure. The complexity is $O(h)$ assuming constant-time $rank$ (Section 6.1).

Algorithm GV-CSA-lookup(i, ℓ)
(1) **if** $\ell = h$ **then return** $A_h[i]$;
(2) **if** $B^\ell[i] = 1$
(3) **then return** $2 \cdot$ GV-CSA-lookup($rank_1(B^\ell, i)$, $\ell + 1$);
(4) **else return** GV-CSA-lookup($\Psi_\ell[rank_0(B^\ell, i)]$, ℓ) $- 1$;

Alg. 2: Obtaining $A[i]$ from GV-CSA recursive structure with h levels. It is invoked as GV-CSA-lookup(i,0).

It is convenient to use $h = \lceil \log \log n \rceil$, so that the $n/2^h$ entries of A_h, each of which requires $O(\log n)$ bits, take overall $O(n)$ bits. All the B^ℓ arrays add up at most $2n$ bits (as their length is halved from each level to the next), and their additional *rank* structures add $o(n)$ extra bits (Section 6.1). The only remaining problem is how to represent the Ψ_ℓ arrays.

For a compact representation of Ψ_0, we recall that Ψ is increasing within the area of A that points to suffixes starting with the same character (Lemma 1). Although Grossi and Vitter [2000] do not detail how to use this property to represent Ψ in little space, an elegant solution is given in later work by Sadakane [2000, 2003]. Essentially, Sadakane shows that Ψ can be encoded differentially ($\Psi(i+1) - \Psi(i)$) within the areas where it is increasing, using Elias-δ coding [Elias 1975; Witten et al. 1999] (recall Section 4.5). The number of bits this representation requires is $nH_0 + O(n \log \log \sigma)$. For Ψ_0, since only odd text positions are considered, the result is the same as if we had a text $T'_{1,n/2}$ formed by bigrams of T, $T'_i = T_{2i-1,2i}$. Since the zero-order entropy of T taken as a sequence of 2^ℓ-grams is $H_0^{(2^\ell)} \le 2^\ell H_0$ [Sadakane 2003], Ψ_0 requires $|T'|H_0^{(2)} + O(|T'| \log\log(\sigma^2)) \le (n/2)(2H_0) + O((n/2)(1 + \log\log\sigma))$. In general, Ψ_ℓ requires at most $(n/2^\ell)(2^\ell H_0) + O((n/2^\ell)(\ell + \log\log\sigma)) = nH_0 + O(n\ell/2^\ell) + O((n \log\log\sigma)/2^\ell)$ bits. Overall, the h levels require $hnH_0 + O(n \log\log\sigma)$ bits.

In order to access the entries of these compressed Ψ_ℓ arrays in constant time [Sadakane 2003], absolute Ψ_ℓ values are inserted at entry boundaries every $\Theta(\log n)$ bits (not $\Theta(\log n)$ entries, as in the simplified solution of Section 4.5), so this adds $O(n)$ bits. To extract an arbitrary position $\Psi_\ell[i]$, we go to the nearest absolute sample before i and sequentially advance summing up differences until reaching position i. To know which is the nearest sample position preceding i one can have bit arrays $\texttt{posSamp}_{1,n}$ telling which entries are sampled, and $\texttt{startSamp}_{1,|\Psi_\ell|}$ marking the positions in the bit stream representing Ψ_ℓ where absolute samples start. Then the last sampled position before i is $i' = select_1(\texttt{posSamp}, rank_1(\texttt{posSamp}, i))$. The absolute value of $\Psi_\ell[i']$ starts at bit position $select_1(\texttt{startSamp}, rank_1(\texttt{posSamp}, i))$ in Ψ_ℓ. By using the techniques of Section 6.2, these two arrays require $O((n + |\Psi_\ell|) \log\log n / \log n)$ extra bits of space, which is negligible.

We must also be able to process all the bits between two samples in constant time. By maintaining a precomputed table with the total number of differences encoded inside every possible chunk of $\frac{\log n}{2}$ bits, we can process each such chunk in constant time, so the $\Theta(\log n)$ bits of differences can also be processed in constant time. The size of that table is only $O(\sqrt{n}\ \log^2 n) = o(n)$ bits. Note the similarity with the other four-Russians technique for constant time *rank* (Section 6.1).

What we have, overall, is a structure using $nH_0 \log\log n + O(n \log\log\sigma)$ bits of space, which encodes A and permits retrieving $A[i]$ in $O(\log\log n)$ time.

A tradeoff with $\frac{1}{\epsilon}\, nH_0 + O(n\log\log\sigma)$ bits of space and $O(\log^{\epsilon} n)$ retrieval time, for any constant $\epsilon > 0$, can be obtained as follows. Given the $h = \lceil\log\log n\rceil$ levels, we only keep three levels: 0, $\lfloor h/2\rfloor$, and h. Bit vectors B^0 and $B^{\lfloor h/2\rfloor}$ indicate which entries are represented in levels $\lfloor h/2\rfloor$ and h, respectively. The space for Ψ_0, $\Psi_{\lfloor h/2\rfloor}$, and A_h, is at most $2nH_0 + O(n\log\log\sigma)$ bits. However, we cannot move from one level to the next in constant time. We must use Ψ_ℓ several times until reaching an entry that is sampled at the next level. The number of times we must use Ψ_ℓ is at most $2^{h/2} = O(\sqrt{\log n})$. If, instead of 3 levels, we use a constant number $1 + 1/\epsilon$ of levels 0, $h\epsilon$, $2h\epsilon$, …, h, the time is $O(\log^{\epsilon} n)$. By applying the usual algorithms over this representation of A we get the following results.

Theorem 10 [Grossi and Vitter 2000; Sadakane 2000] *The* Compressed Suffix Array *of Grossi and Vitter (*GV-CSA*) offers the following space/time tradeoffs.*

Space in bits	$nH_0\log\log n + O(n\log\log\sigma) + n\log\sigma$
Time to count	$O(m\log n\log\log n)$
Time to locate	$O(\log\log n)$
Time to display ℓ chars	$O(\ell)$ (text is available)
Space in bits	$\frac{1}{\epsilon}\, nH_0 + O(n\log\log\sigma) + n\log\sigma$
Time to count	$O(m\log^{1+\epsilon} n)$
Time to locate	$O(\log^{\epsilon} n)$
Time to display ℓ chars	$O(\ell)$ (text is available)
Conditions	$0 < \epsilon \leq 1$ *is an arbitrary constant*

We have described the solution of Sadakane [2000, 2003] to represent Ψ in little space and constant access time. The solution of the original authors has just appeared [Grossi and Vitter 2006] and it is slightly different. They also use the fact that Ψ is piecewise increasing in a different way, achieving $\frac{1}{2}n\log\sigma$ bits at each level instead of nH_0. Furthermore, they take T as a binary string of $n\log\sigma$ bits, which yields essentially $n\log\sigma\log\log_\sigma n$ bits for the first version of Theorem 10 and $\frac{1}{\epsilon}\, n\log\sigma$ bits for the second version. They actually use $h = \lceil\log\log_\sigma n\rceil$, which adds up $n\log\sigma$ extra space for A_h and slightly reduces the time to access $A[i]$ in the first variant of the above theorem to $O(h) = O(\log\log_\sigma n)$.

Grossi and Vitter [2000, 2006] show how the *occ* occurrences can be located more efficiently in batch when m is large enough. They also show how to modify a compressed suffix tree [Munro et al. 2001] so as to obtain $O(m/\log_\sigma n + \log^{\epsilon} n)$ search time, for any constant $0 < \epsilon < 1$, using $O(n\log\sigma)$ bits of space. This is obtained by modifying the compressed suffix tree [Munro et al. 2001] in two ways: First, using perfect hashing to allow traversing $O(\log_\sigma n)$ tree nodes downwards in one step, and second, replacing the suffix array required by the compressed suffix tree with the GV-CSA. We do not provide details because in this paper we focus on indexes taking $o(n\log\sigma)$ bits. In this sense, we are not interested in the GV-CSA by itself, but as a predecessor of other self-indexes that appeared later.

A generalization of this structure (but still not a self-index) is presented by Rao [2002], where they index a binary text using $O(nh\log^{1/h} n)$ bits and retrieve $A[i]$ in $O(h)$ time, for any $1 \leq h \leq \log\log n$.

8. TURNING COMPRESSED SUFFIX ARRAYS INTO SELF-INDEXES

Further development of the techniques of Section 7 lead to self-indexes, which can operate without the text. The first index in this line was the *Compressed Suffix Array* of Sadakane [2000, 2002, 2003] (Sad-CSA). This was followed by the *Compressed Suffix Array* of Grossi, Gupta, and Vitter [2003, 2004] (GGV-CSA), and by the *Compressed Compact Suffix Array* of Mäkinen and Navarro [2004a] (MN-CCSA). The latter, which builds on the Mak-CSA, is not covered here.

8.1 Sad-CSA: Sadakane's Compressed Suffix Array

Sadakane [2000, 2003] showed how the GV-CSA can be converted into a self-index, and at the same time optimized it in several ways. The resulting index was also called *Compressed Suffix Array* and will be referred to as Sad-CSA in this paper.

The Sad-CSA does not give, as the GV-CSA, direct access to $A[i]$, but rather to any prefix of $T_{A[i],n}$. This still suffices to use the search algorithm of Alg. 1. The Sad-CSA represents both A and T using the full function Ψ (Definition 11), and a few extra structures (recall Section 4.4).

Imagine we wish to compare P against $T_{A[i],n}$. For the binary search, we need to extract enough characters from $T_{A[i],n}$ so that its lexicographical relation to P is clear. Since $T_{A[i]}$ is the first character of the suffix pointed to by $A[i]$, we have $T_{A[i]} = F_i$ in Fig. 8. Once we determine $T_{A[i]} = c$ in this way, we need to obtain the next character, $T_{A[i]+1}$. But $T_{A[i]+1} = T_{A[\Psi(i)]}$, so we simply move to $i' = \Psi(i)$ and keep extracting characters with the same method, as long as necessary. Note that at most $|P| = m$ characters suffice to decide a comparison with P.

In order to quickly find $c = T_{A[i]}$, we store a bit vector $\mathtt{newF}_{1,n}$, so that $\mathtt{newF}_i = 1$ iff $i = 1$ or $F_i \neq F_{i-1}$, and a string `charT` where the (at most σ) distinct characters of T are concatenated in alphabetical order. Then we have $c = \mathtt{charT}[rank_1(\mathtt{newF}, i)]$, which can be computed in constant time using only $n + o(n)$ bits for `newF` (Section 6.1) and $\sigma \log \sigma$ bits for `charT`.

Fig. 12 illustrates. To obtain $T_{A[11],n}$ we see that $\mathtt{charT}[rank_1(\mathtt{newF}, 11)] = \mathtt{charT}[3] = \texttt{"a"}$. Then we move to $19 = \Psi(11)$, so that the second character is $\mathtt{charT}[rank_1(\mathtt{newF}, 19)] = \mathtt{charT}[6] = \texttt{"l"}$. We now move to $9 = \Psi(19)$ and get the third character $\mathtt{charT}[rank_1(\mathtt{newF}, 9)] = \mathtt{charT}[3] = \texttt{"a"}$, and so on. Note that we are, implicitly, walking the text in forward direction. Note also that we do not know where we are in the text: We never know $A[i]$, just $T_{A[i],n}$.

Thus the Sad-CSA implements the binary search in $O(m \log n)$ worst-case time, which is better than in the GV-CSA structure. Alg. 3 gives the pseudocode to compare P against a suffix of T.

Up to now we have used $n + o(n) + \sigma \log \sigma$ bits of space for `newF` and `charT`, plus the representation for Ψ described in Section 7.1, $nH_0 + O(n \log \log \sigma)$ bits. Note that, since the Sad-CSA does not give direct access to $A[i]$, it needs more structures to solve a locating query. That is, although the index knows that the answers are in $A[sp, ep]$ and thus that there are $occ = ep - sp + 1$ answers, it does not have enough information to know the text positions pointed to by $A[i]$, $sp \leq i \leq ep$. For this sake, the Sad-CSA includes the hierarchical GV-CSA structure (without the text and with Ψ instead of Ψ_0, as we already have the more complete Ψ). Let us choose, from Theorem 10, the version requiring $\frac{1}{\epsilon} nH_0 + O(n \log \log \sigma)$ bits of

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A =	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18
Ψ =	10	7	11	17	1	3	4	14	15	18	19	20	21	12	13	5	6	8	9	2	16
newF =	1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0
T =	a	l	a	b	a	r	_	a	_	l	a	_	a	l	a	b	a	r	d	a	$

charT = $_abdlr

Fig. 12. The main components of the SAD-CSA structure, for the text `"alabar a la alabarda$"`. We show T and A for illustrative purposes, yet these are not stored in the structure.

Algorithm Sad-CSA-compare(P, m, i, Ψ, `newF`, `charT`)
$j \leftarrow 1$;
do $c \leftarrow$ `charT`$[rank_1($`newF`$, i)]$;
  if $P_j < c$ **then return** "<";
  if $P_j > c$ **then return** ">";
  $i \leftarrow \Psi(i)$;
  $j \leftarrow j + 1$;
while $j \leq m$;
return "=";

Alg. 3: Comparing P against $T_{A[i],n}$ using Ψ, `newF`, and `charT`.

space and computing $A[i]$ in $O(\log^\epsilon n)$ time, for any constant $0 < \epsilon < 1$.

The remaining functionality a self-index must provide is to retrieve $T_{l,r}$ without having access to T. We already know how to retrieve any prefix of $T_{A[i],n}$ given i. Note that, if we had the inverse of permutation A, $A^{-1}[j]$, we could first find $i = A^{-1}[l]$ and then retrieve $T_{l,r} = T_{A[i],A[i]+(r-l)}$ in $O(r - l + 1)$ additional time.

Sadakane notices that the same hierarchical GV-CSA structure can serve to compute $A^{-1}[j]$ in time $O(\log^\epsilon n)$, provided we also store explicitly A_h^{-1} in the last level. The reason is that $A^{-1}[j] = \Psi(A^{-1}[j-1])$, which is easy to verify. Iterating, $A^{-1}[j] = \Psi^k(A^{-1}[j-k])$. Hence, to compute $A^{-1}[j] = A_0^{-1}[j]$ we take the largest $j' \leq j$ that is represented in the next level $h\epsilon$ (that is, j' is the largest multiple of $2^{h\epsilon}$ not larger than j), and obtain $i' = A_0^{-1}[j'] = select_1(B^0, A_{h\epsilon}^{-1}[\lfloor j/(h\epsilon) \rfloor])$ recursively (or directly when we reach level h). Now we apply Ψ successively over i' to obtain $A_0^{-1}[j] = \Psi_0^{j-j'}[i']$. This takes $O((1 + \frac{1}{\epsilon})2^{h\epsilon}) = O(\log^\epsilon n)$ time.

Theorem 11 [Sadakane 2003] *The* Compressed Suffix Array *of Sadakane (*SAD-CSA*) offers the following space/time tradeoff.*

Space in bits	$\frac{1}{\epsilon} nH_0 + O(n \log\log\sigma) + \sigma\log\sigma$
Time to count	$O(m \log n)$
Time to locate	$O(\log^\epsilon n)$
Time to display ℓ chars	$O(\ell + \log^\epsilon n)$
Conditions	$0 < \epsilon \leq 1$ *is an arbitrary constant*

We note that the combination absolute samples and the four-Russian technique to access Ψ works with many other compression methods. In particular, if we compress runs in Ψ (Definition 12) with run-length compression (see Section 9.5), we can achieve $nH_k(T)(\log\sigma + \log\log n) + O(n)$ bits of space, for $k \leq \log_\sigma n - \omega(1)$ (recall Theorem 5), while retaining the same search times [Mäkinen and Navarro 2004b]. This tradeoff is the same of the MN-CCSA [Mäkinen and Navarro 2004a]. We show an even better tradeoff, unnoticed up to now, at the end of Section 8.2.

In practice. The implementation of SAD-CSA differs in several aspects from its theoretical description. First, it does not implement the inverse suffix array to locate occurrences. Rather, it samples A at regular intervals of length D, explicitly storing $A[i \cdot D]$ for all i. In order to obtain $A[j]$, we compute Ψ repeatedly over j until obtaining a value $j' = \Psi^r(j)$ that is a multiple of D. Then $A[j] = A[j'] - r$. Similarly, constant access to Ψ is not provided. Rather, absolute Ψ values are sampled every D' positions. To obtain $\Psi(i)$, we start from its closest previous absolute sample and decompress the differential encoding until position i. Finally, instead of the classical suffix array searching, backward searching is used [Ferragina and Manzini 2000; Sadakane 2002]. This avoids any need to obtain text substrings, and it is described in Section 9.2. Thus, D and D' give practical tradeoffs between index space and search time.

8.2 GGV-CSA: Grossi, Gupta, and Vitter's Compressed Suffix Array

The *Compressed Suffix Array* of Grossi, Gupta, and Vitter [2003, 2004] (GGV-CSA) is a self-index whose space usage depends on the k-th order entropy of T. It is an evolution over the GV-CSA and the SAD-CSA, based on a new representation of Ψ that requires essentially nH_k bits rather than nH_0, mainly using Theorem 4.

Consider Figs. 3 (page 9) and 12, and the text context $s =$ `"la"`. Its occurrences are pointed from $A[17,19] = \{10, 2, 14\}$. The Ψ values that point to that interval are $\Psi(4) = 17$, $\Psi(10) = 18$, and $\Psi(11) = 19$. The first corresponds to character $T_{A[4]} =$ `" "` preceding `"la"`, while the other two correspond to `"a"` preceding `"la"`.

Fig. 13 illustrates how the sequence of Ψ values is partitioned into *lists* ($\Psi(i)$ belongs to list $T_{A[i]}$, that is, the character preceding position $\Psi(i)$ in T) and *contexts* ($\Psi(i)$ belongs to context $T_{A[\Psi(i)],A[\Psi(i)]+k-1} = T_{A[i]+1,A[i]+k}$, that is, the text starting at $\Psi(i)$ in T). The Ψ values are increasing within a cell. Seen another way, suffix array position j ($= \Psi(i)$) belongs to list $T_{A[j]-1}$ and context $T_{A[j],A[j]+k-1}$. For example, with $k = 2$, suffix array position 17 ($= \Psi(4)$) points to $T_{A[17],n} = T_{A[4]+1,n} = T_{10,n}$, so it belongs to list $T_9 = T_{A[17]-1} = T_{A[4]} =$ `" "` and context $T_{10,11} = T_{A[17],A[17]+1} = T_{A[4]+1,A[4]+2} =$ `"la"`, whereas position 18 ($= \Psi(10)$) belongs to list $T_{A[10]} = T_1 =$ `"a"` and context $T_{A[18],A[18]+1} = T_{2,3} =$ `"la"`.

The duality we have pointed out is central to understand this index: The table of lists and contexts is arranging the numbers in the interval $[1, n]$. Those numbers can be regarded in two ways. The main one is that they are the values $\Psi(1)$, $\Psi(2)$, ..., $\Psi(n)$, that is, we are storing vector Ψ. The secondary one is that these Ψ values are indexes to array A, thus we are storing the indexes of A.

If we sort the sequence of values in the table by list, and by context inside each list, then we have the entries $\Psi(i)$ sorted by $T_{A[i],A[i]+k}$, that is, in increasing order of i. Thus we recover the original sequence Ψ in good order if we read the table

Context	List `"$"`	List `" "`	List `"a"`	List `"b"`	List `"d"`	List `"l"`	List `"r"`
`"$"`			1,				
`" a"`			3,				2,
`" l"`			4,				
`"a$"`					5,		
`"a "`		7,				6,	
`"ab"`	10,					8, 9,	
`"al"`		11,					
`"ar"`				12, 13,			
`"ba"`			14, 15,				
`"da"`							16,
`"la"`		17,	18, 19,				
`"r "`			20,				
`"rd"`			21,				

Fig. 13. Partition of Ψ into lists (columns) and contexts (rows) in the GGV-CSA, for the text `"alabar a la alabarda$"`. Array Ψ is read columnwise left to right, each column top to bottom.

columnwise (left to right), and each column top to bottom (as we ordered lists and contexts lexicographically). The order within each cell (same $T_{A[i],A[i]+k}$) is also correct because Ψ is increasing in there, and we store the values that way within cells. Actually, the Ψ values are increasing along any whole column, as $\Psi(i)$ values are increasing as long as $T_{A[i]}$ (the column) does not change (Lemma 1).

Now regard the numbers as indexes j $(= \Psi(i))$ of A. The row where each j value lies corresponds to the context its pointed suffix starts with, $s = T_{A[j],A[j]+k-1} = T_{A[\Psi(i)],A[\Psi(i)]+k-1}$. Thus, the j values found in a row form a contiguous subinterval of $[1, n]$ (indexes of A). Each cell in the row corresponds to a different character preceding the context (that is, to a different column).

If we identify each j value in the row with the character of the column it belongs to $(T_{A[j]-1})$, then the set of all characters form precisely T^s (Definition 8), of length $n_s = |T^s|$. Thus, if we manage to encode each row in $n_s\, H_0(T^s)$ bits, we will have $nH_k(T)$ bits overall (recall Eq. (1) and Theorem 4). In our previous example, considering context $s =$ `"la"`, we have to represent all the j values that lie inside that context ($[17, 19]$) in space proportional to the zero-order entropy of the characters $T_{A[j]-1}$, $j \in [17, 19]$, that is, $T^s = T^{bwt}_{17,19} =$ `" aa"`.

To obtain $\Psi(i)$ from this table we first need to determine the row and column i belongs to, and then the position inside that cell. To know the column c, bitmap `newF` of Fig. 12 (G_k in [Grossi et al. 2003]) suffices, as $c = rank_1(\texttt{newF}, i)$ (for simplicity we are identifying characters and column numbers). Using the techniques of Section 6.2, `newF` can be represented in $nH_0(\texttt{newF}) + o(n) \le \sigma \log n + o(n)$ bits (as it has at most σ bits set, recall the end of Section 5.1), so that it answers $rank_1$ queries in constant time. The relative position of i inside column c is $i' = i - select_1(\texttt{newF}, c) + 1$. In the example, to retrieve $\Psi(10) = 18$ (thus $i = 10$), we find $c = rank_1(\texttt{newF}, 10) = 3$ (third column in the table, symbol `"a"`). Inside the column, we want the 6th value, because $i' = 10 - select_1(\texttt{newF}, 3) + 1 = 6$.

A similar technique gives the right cell within the column. Two bit arrays $\texttt{newRow}_c$ and $\texttt{isRow}_c$ (L^y_k and b^y_k in [Grossi et al. 2003]) are maintained for each column c. $\texttt{newRow}_c$ is aligned to the area of Ψ that belongs to column c. It has a 1-bit every time we enter a new row as we read the values in column c. In our example, $\texttt{newRow}_\texttt{a} = 111101011$, which is aligned to $\Psi[5, 13]$. $\texttt{isRow}_c$, instead, stores one bit per context indicating whether the row for that context is nonempty in column c. In our example, $\texttt{isRow}_\texttt{a} = 1110000010111$.

It is easy to see that the relative index i' within column c corresponds to the r'-th nonempty cell of column c, where $r' = rank_1(\texttt{newRow}_c, i')$, and the r' nonempty cell has global row number $r = select_1(\texttt{isRow}_c, r')$. Finally, the position we want inside the r'-th nonempty cell is $p = i' - select_1(\texttt{newRow}_c, r') + 1$. In our example, the cell is at the 5th nonempty row, as $r' = rank_1(\texttt{newRow}_\texttt{a}, 6) = 5$. Its global row number is $r = select_1(\texttt{isRow}_\texttt{a}, 5) = 11$. Its position within the cell is the first, as $p = 6 - select_1(\texttt{newRow}_\texttt{a}, 5) + 1 = 1$.

Using again Section 6.2, each $\texttt{newRow}_c$ can be stored in $n_c\, H_0(\texttt{newRow}_c) + o(n_c)$ bits (note that n_c is the number of elements in column c), and answer those queries in constant time. As there are at most σ^k bits set in $\texttt{newRow}_c$, the space is at most $\sigma^k \log(n_c) + o(n_c)$ bits. Added over all columns, this is at most $\sigma^{k+1} \log n + o(n)$ (Grossi et al. [2003] concatenate all $\texttt{newRow}_c$ vectors for technical reasons we omit). In turn, $\texttt{isRow}_c$ vectors add up $\sigma^{k+1}(1 + o(1))$ bits using Section 6.1.

We also need to know which is the range of suffix array indexes j handled by the row. For example, for $s =$ "la", row 11, we must know that this context corresponds to the suffix array interval $[17, 19]$. We store a bitmap newCtx (F_k in [Grossi et al. 2003]) whose positions are aligned to A, storing a 1 each time a context change occurs while traversing A. This is the global version of $\texttt{newRow}_c$ bit vectors (which record context changes within column c). In our example, $\texttt{newCtx} = 110111010011010110011$. If we know we are in global row r, then $select_1(\texttt{newCtx}, r)$ tells the first element in the interval handled by row r. In our example, $r = 11$ and $select_1(\texttt{newCtx}, 11) = 17$. We will add this value to the result we obtain inside our cell. Using Section 6.2 once more, newCtx requires $\sigma^k \log n + o(n)$ bits.

The final piece is to obtain the p-th element of cell (r, c). At this point there are different choices. One, leading to Theorem 12, is to store a bitmap $B^{r,c}$ (Z in [Grossi et al. 2003]) for each cell (r, c), indicating which elements of the row interval belong to the cell. This is necessary because any permutation of the row interval can arise among the cells (e.g., see row "ab"). In our example, for row 11, the interval is $[17, 19]$, and we have $B^{11,\text{" "}} = 100$ and $B^{11,\texttt{a}} = 011$ (zeros and ones can be interleaved in general). With $select_1(B^{r,c}, p)$ we obtain the offset of our element in the row interval. Therefore, we finally have $\Psi(i) = select_1(\texttt{newCtx}, r) + select_1(B^{r,c}, p) - 1$. In our example, $select_1(011, 1) = 2$, and thus $\Psi(10) = 17 + 2 - 1 = 18$. Alg. 4 gives the pseudocode.

Algorithm GGV-CSA-Ψ(i, newF, newRow, isRow, newCtx, B)
(1) $c \leftarrow rank_1(\texttt{newF}, i)$;
(2) $i' \leftarrow i - select_1(\texttt{newF}, c) + 1$;
(3) $r' \leftarrow rank_1(\texttt{newRow}_c, i')$;
(4) $p \leftarrow i' - select_1(\texttt{newRow}_c, r') + 1$;
(5) $r \leftarrow select_1(\texttt{isRow}_c, r)$;
(6) **return** $select_1(\texttt{newCtx}, r) + select_1(B^{r,c}, p) - 1$;

Alg. 4: Computing $\Psi(i)$ using the GGV-CSA.

We consider now the space required by the bit vectors B. Using again the techniques of Section 6.2, those can be stored in $|B|H_0(B) + O(|B| \log\log |B| / \log |B|)$ bits. Note that $|B^{r,c}| = n_s$ for each cell in row r corresponding to context s. Summed over all the lists of row r, this is $n_s\, H_0(T^s) + n_s \log e + O(n_s \log\log n_s / \log n_s)$, and added over all the contexts s, we get $nH_k(T) + n \log e + O(n \log\log n / \log(n/\sigma^k))$ (Eq. (1) and Theorem 4). To see that the sum of the $H_0(B)$ entropies over the different columns adds up to $H_0(T^s)$, let us call b_c the number of entries in cell (r, c). Then, recalling the end of Section 5.1, $H_0(B) \le b_c \log \frac{b}{b_c} + b_c \log e$, which add up $\sum_{c\in\Sigma}(b_c \log \frac{b}{b_c} + b_c \log e)$. But b_c is also the number of occurrences of c in T^s, so the sum is $n_s\, H_0(T^s) + n_s \log e$.

Let us assume $k \le \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$. This ensures that our overall space requirement up to now is $nH_k(T) + n \log e + o(n)$.

However, we are not representing Ψ, but Ψ_ℓ for $0 \le \ell \le h$ (Section 7.1). The structure above works verbatim for Ψ_0, but it also can be used for any level ℓ. The difference is that at level ℓ there are not σ lists, but rather σ^{2^ℓ}. Recalling the space analysis in Section 7.1), the space requirement at level ℓ turns out to be $nH_k(T) + (n/2^\ell) \log e + o(n)$ bits as long as $k + 2^\ell \le \alpha \log_\sigma n$. For this we need two conditions: (1) $k \le \alpha' \log_\sigma n$ for some constant $0 < \alpha' < 1$, so we can choose any $\alpha \in (\alpha', 1)$; (2) to stop the recursive structure at level $h' = \log((\alpha - \alpha') \log_\sigma n) = \Theta(\log\log_\sigma n)$, so that $2^\ell \le (\alpha - \alpha') \log_\sigma n$ when $\ell \le h'$. The levels between h' and $h = \lceil \log\log n \rceil$ (where we store A_h explicitly) must be omitted, and this means that we must jump directly from level h' to level h, just as when we used a constant number of levels in Theorem 10. The number of steps for that jump is $2^{h-h'} = O(\log \sigma)$.

As we can access each Ψ_ℓ in constant time, we first pay $O(h')$ time to reach level h' and then pay $O(\log \sigma)$ to reach level h. This is $O(\log\log_\sigma n + \log \sigma) = O(\log\log n + \log \sigma)$ time to access $A[i]$. For the space, we have h' levels taking $nH_k(T) + (n/2^\ell) \log e + o(n)$ each, plus the final level containing A and its inverse. This yields the first result of Grossi et al. [2003].

Theorem 12 [Grossi et al. 2003] *The* Compressed Suffix Array *of Grossi, Gupta and Vitter (*GGV-CSA*) offers the following space/time tradeoffs.*

Space in bits	$nH_k \log\log_\sigma n + 2(\log e + 1)n + o(n)$
Time to count	$O(\log n(m/\log_\sigma n + \log\log n + \log \sigma))$
Time to locate	$O(\log\log n + \log \sigma)$
Time to display ℓ chars	$O(\ell/\log_\sigma n + \log\log n + \log \sigma)$
Space in bits	$\frac{1}{\epsilon}\, nH_k + 2(\log e + 1)n + o(n)$
Time to count	$O(\log n(m/\log_\sigma n + \log^\epsilon n + \log \sigma))$
Time to locate	$O(\log^\epsilon n + \log \sigma)$
Time to display ℓ chars	$O(\ell/\log_\sigma n + \log^\epsilon n + \log \sigma)$
Conditions for all	$0 < \epsilon \le 1$ *is an arbitrary constant;* $k \le \alpha \log_\sigma n$*, for some constant* $0 < \alpha < 1$

Note that, compared to Theorem 11, m and ℓ are now divided by $\log_\sigma n$. This is because Grossi et al. [2003] note that the process carried out by the Sad-CSA to extract the text using Ψ character by character can actually be carried out at $\Psi_{h'}$, where $2^{h'} = O(\log_\sigma n)$ characters are obtained in each step. Thus the counting time is $O(m \log \sigma + \text{polylog}(n))$. The $O(\log n)$ factor multiplying m in previous approaches becomes now $O(\log \sigma)$, although new polylogarithmic terms in n appear. On the other hand, the version using $\frac{1}{\epsilon} nH_k$ bits is obtained just as with the GV-CSA, using a constant number $1 + 1/\epsilon$ of levels in $[0, h']$.

We observe that there is still an $O(n)$ term in the space complexity, which is a consequence of representing each $B^{r,c}$ individually. An alternative is to represent the whole row using a wavelet tree (Section 6.3, invented by Grossi et al. [2003] for this purpose). The idea is that, for each row of context s, we encode sequence T^s with the binary wavelet tree of Theorem 8. In our example, for row $s =$ `"la"`, we encode $T^s =$ `" aa"`. In order to retrieve element p from cell (r, c), we just compute $select_c(T^s, p)$, adding it to $select_1(\texttt{newCtx}, r) - 1$ to return the value in the final line of Alg. 4. In our example, $select_{\texttt{a}}(\texttt{" aa"}, 1) = 2$ replaces $select_1(011, 1) = 2$.

The binary wavelet tree requires $n_s\, H_0(T^s) + o(n_s \log \sigma)$ bits of space and answers the queries in $O(\log \sigma)$ time. Adding over all the contexts s we get $nH_k(T) + o(n \log \sigma)$ bits, for $k \le \alpha \log_\sigma n$. In exchange, the time increases by an $O(\log \sigma)$ factor. In level ℓ, the $\log \sigma$ terms become $\log(\sigma^{2^\ell}) = 2^\ell \log \sigma$. To ensure the extra space stays $o(n \log \sigma)$ we must ensure $2^{h'} = o(\log n / \log\log n)$, for example, $h' \le (\alpha - \alpha') \log\log_\sigma n - 2 \log\log\log n$. Yet, we still have another space problem, namely the extra $O(n)$ bits due to storing A_h and A_h^{-1}. These are converted into $o(n \log \sigma)$ by setting, for example, $h = \log\log_\sigma n + \log\log\log n$.

The search time added over $1+1/\epsilon$ levels in $0 \le \ell \le h'$ is $O(\log^{1+\epsilon} n/(\log\log n)^2)$, while the time to move from level h' to h is $O(2^{h-h'} \cdot 2^{h'} \log \sigma) = O(\log n \log\log n)$.

Theorem 13 **[Grossi et al. 2003]** *The* Compressed Suffix Array *of Grossi, Gupta and Vitter (*GGV-CSA*) offers the following space/time tradeoffs.*

Space in bits	$\frac{1}{\epsilon} nH_k + o(n \log \sigma)$
Time to count	$O(m \log \sigma + \log^{2+\epsilon} n)$
Time to locate	$O(\log^{1+\epsilon} n)$
Time to display ℓ chars	$O(\ell / \log_\sigma n + \log^{1+\epsilon} n)$
Conditions	$0 < \epsilon \le 1$ *is an arbitrary constant;*
	$k \le \alpha \log_\sigma n$, *for some constant* $0 < \alpha < 1$

More complicated tradeoffs are given by Grossi et al. [2003]. The most relevant ones obtain, roughly, $O(m/\log_\sigma n + \log^{\frac{2\epsilon}{1-\epsilon}} n)$ counting time with $\frac{1}{\epsilon} nH_k + o(n \log \sigma)$ bits of space, for any $0 < \epsilon < 1/3$; or $O(m \log \sigma + \log^4 n)$ counting time with almost optimal $nH_k + o(n \log \sigma)$ space.

In practice. Some experiments and practical considerations are given by Grossi et al. [2004]. They show that bit vectors B can be represented using run-length encoding and then Elias-γ [Elias 1975; Witten et al. 1999], so that they take at most $2|B|H_0(B)$ bits (and they may take less if the bits are not uniformly distributed). Note that this result was partially foreseen by Sadakane [2000, 2003] to achieve zero-order encoding of Ψ in the Sad-CSA (Section 7.1). They do not explain how to do *rank* and *select* in constant time on this representation, but in [Grossi and Vitter 2006] they explore binary-searchable gap encodings as a practical alternative.

An interesting result of Grossi et al. [2004] is that, since the sum of all the γ-encodings across all the cells adds up $2nH_k(T)$ bits, we could use the same encoding to code each column in Fig. 13 as a whole. The values within a column are increasing. The total space for this representation is that of the γ-encoding inside the cells (which overall amounts to $2nH_k$ bits) plus that of the γ-encoding of the jumps between cells. The latter is $o(n)$ as long as $k \leq \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$. Thus, we obtain $2nH_k + o(n)$ bits. Note, and this is the key part, that the sequence of differences we have to represent is the same no matter how the values are split along the rows. That is, the sequence (and its space) is the same Ψ independently of how long the contexts are. Therefore, this encoding achieves $2nH_k + o(n)$ *implicitly and simultaneously* for any $k \leq \alpha \log_\sigma n$. This is in contrast with their original work [Grossi et al. 2003], where k had to be chosen at indexing time. Interestingly, this also shows that the Elias-δ representation of the SAD-CSA (where in fact a column-wise differential representation is used for Ψ) actually requires $nH_k + O(n \log\log\sigma)$ bits of space, improving the analysis by Sadakane [2000, 2003] (contrast with the other nH_k-like solution at the end of Section 8.1).

9. BACKWARD SEARCHING AND THE FM-INDEX FAMILY

Backward searching is a completely different approach to searching using suffix arrays. It matches particularly well with the BWT (Section 5.3), but it can also be applied with compressed suffix arrays based on the Ψ function, using the fact that Ψ and LF are the inverse of each other (Lemma 4). The first exponent of this idea was the FM-Index of Ferragina and Manzini [2000], and many others followed. We first present the idea and then describe its different realizations (see also Section 4.1).

9.1 The Backward Search Concept

Consider searching for P in A as follows. We first determine the range $[sp_m, ep_m]$ in A of suffixes starting with P_m. Since P_m is a single character, function C of Lemma 3 can be used to determine $[sp_m, ep_m] = [C(P_m)+1, C(P_m+1)]$ (we use $c+1$ to denote the character that follows c in Σ). Now, given $[sp_m, ep_m]$, we want to compute $[sp_{m-1}, ep_{m-1}]$, the interval of A corresponding to suffixes starting with $P_{m-1,m}$. This is of course a subinterval of $[C(P_{m-1})+1, C(P_{m-1}+1)]$. In the general case, we know the interval $[sp_{i+1}, ep_{i+1}]$ of A corresponding to suffixes that start with $P_{i+1,m}$ and want $[sp_i, ep_i]$, which is a subinterval of $[C(P_i)+1, C(P_i+1)]$. At the end, $[sp_1, ep_1]$ is the answer for P.

The LF-mapping (Definition 14) is the key to obtain $[sp_i, ep_i]$ from $[sp_{i+1}, ep_{i+1}]$. We know that all the occurrences of P_i in $L[sp_{i+1}, ep_{i+1}]$ appear contiguously in F, and they preserve their relative order. Let b and e be the first and last position in $[sp_{i+1}, ep_{i+1}]$ where $L_b = L_e = P_i$. Then, $LF(b)$ and $LF(e)$ are the first and last rows of M that start with $P_{i,m}$. Recall from Lemma 3 that $LF(b) = C(L_b) + Occ(L_b, b)$ and $LF(e) = C(L_e) + Occ(L_e, e)$. The problem is that we do not know b and e. Yet, this is not necessary. Since b is the position of the first occurrence of P_i in $L[sp_{i+1}, ep_{i+1}]$, it follows that $Occ(L_b, b) = Occ(P_i, b) = Occ(P_i, sp_{i+1} - 1) + 1$. Likewise, $Occ(L_e, e) = Occ(P_i, e) = Occ(P_i, ep_{i+1})$ because e is the last occurrence of P_i in $L[sp_{i+1}, ep_{i+1}]$. The final algorithm is rather simple and is shown in Alg. 5.

Function C is implemented as an array, using just $\sigma \log n$ bits. All the different variants of backward searching aim basically at implementing Occ in little time and

```
Algorithm FM-search(P, m, n, C, Occ)
sp ← 1; ep ← n;
for i ← m to 1
    sp ← C(P_i) + Occ(P_i, sp − 1) + 1;
    ep ← C(P_i) + Occ(P_i, ep);
    if sp > ep then return ∅;
    i ← i − 1;
return [sp, ep];
```

Alg. 5: Backward searching for the interval in A of the suffixes that start with $P_{1,m}$.

space. If we achieve constant time for Occ, then the backward search needs just $O(m)$ time, which is better than any compressed suffix array from Section 8.

9.2 Backward Searching using Ψ

Before reviewing the more typical BWT-based implementations, we show that backward searching can be implemented using function Ψ [Sadakane 2002].

Since Ψ and LF are inverse functions, we might binary search values $LF(b)$ and $LF(e)$ using Ψ. Imagine we already know $[sp_{i+1}, ep_{i+1}]$ and $[C(P_i)+1, C(P_i+1)]$. Function Ψ is increasing in the latter interval (Lemma 1). Moreover, $[sp_i, ep_i]$ is the subinterval of $[C(P_i)+1, C(P_i+1)]$ such that $\Psi(j) \in [sp_{i+1}, ep_{i+1}]$ if $j \in [sp_i, ep_i]$. Hence, two binary searches permit obtaining $[sp_i, ep_i]$ in $O(\log n)$ time.

Backward search then completes in $O(m \log n)$ time using the Sad-CSA, just as classical searching. An advantage of backward searching is that it is not necessary at all to obtain text substrings at search time. Sadakane [2002] also shows how the backward search can be implemented in $O(m)$ time if $\sigma = O(\text{polylog}(n))$, essentially using the same idea we present in Section 9.4.

As mentioned at the end of Section 8.1, this is how the Sad-CSA is actually implemented. Also, recall that Ψ is implemented via sampling. The binary searching is performed first over the samples and then completed with a sequential decompression between two samples. If the sampling step is $D = \Theta(\log n)$ (to maintain the space cost of the samples within $O(n)$ bits), then the binary search complexity remains $O(\log n)$ time and the overall search time $O(m \log n)$, even if we do not use four-Russian techniques for fast decompression. If we used normal suffix array searching, we would access $O(m \log n)$ arbitrary positions of Ψ, so the use of four-Russian techniques would be mandatory to avoid a total search cost of $O(m \log^2 n)$.

9.3 FMI: Ferragina and Manzini's Implementation

The first implementation of backward searching was proposed, together with the concept itself, by Ferragina and Manzini [2000]. We call it FMI in this paper.

Ferragina and Manzini [2000] show that Occ can be implemented in constant time, using $O(nH_k)$ bits of space (the FMI was the first structure achieving this space). Essentially, Occ is implemented as the compressed BWT transformed text T^{bwt} plus some directory information.

They compress T^{bwt} by applying move-to-front transform, then run-length compression, and finally a variable-length prefix code. Move-to-front [Bentley et al. 1986] consists of keeping a list of characters ranked by recency, that is, the last character seen is first in the list, then the next-to-last, and so on. Every time we see a new character c, which is at position p in the list, we output p and move c to the beginning of the list. This transform produces small numbers over text zones with few different characters. This is precisely what happens in T^{bwt}. In particular, there tend to appear runs of equal characters in T^{bwt} (precisely, n_{bw} runs, recalling Definition 15), which become runs of 1's after move-to-front. These runs are then captured by the run-length compression. Finally, the prefix code applied is a version of Elias-γ with some provisions for the run lengths. Overall, they show that this representation compresses T^{bwt} to at most $5nH_k(T) + O(\sigma^k \log n)$ bits. This is $5nH_k(T) + o(n \log \sigma)$ for $k \le \log_\sigma(n/\log n) - \omega(1)$.

The directories to answer $Occ(c, i)$ resemble the solutions for *rank* in Sections 6.1 and 6.2. We cut the range $[1, n]$ into *blocks* of length t, grouped in *superblocks* of length t^2. Let us define bit vectors $B^c[i] = 1$ iff $T_i^{bwt} = c$ so that $Occ(c, i) = rank_1(B^c, i)$ (these vectors will not be stored). We use the same $\texttt{superblockrank}_c$ and $\texttt{blockrank}_c$ directories of Section 6.1: if $i = qt + r = q't^2 + r'$, $0 \le r < t$, $0 \le r' < t^2$, then $Occ(c, i) = \texttt{superblockrank}_c[q'] + \texttt{blockrank}_c[q] + rank_1(B^c[qt+1, qt+t], r)$. This final *rank* query is solved by fetching the compressed T^{bwt} stream and processing it using four-Russians techniques. To find the proper block in the compressed T^{bwt} we use directories $\texttt{superblockpos}$ and $\texttt{blockpos}$, as in Section 6.2. All these tables add up $O((n\sigma \log n)/t^2 + (n\sigma \log t)/t)$ bits.

To process the bits of a compressed block in constant time, we must store the state of the move-to-front transformation (that is, the recency rank of characters) at the beginning of each block. This table, $\texttt{mtf}[q]$, requires $O((n\sigma \log \sigma)/t)$ additional bits. The four-Russians table is then $\texttt{smallOcc}[c, o, B, V]$, indexed by a character $c \in \Sigma$, an offset $o \in [1, t]$ inside a block, the compressed content B of a block (a bit stream) whose length is in the worst case $t' = (1 + 2 \log \sigma)t$, and the state V of the move-to-front transformation (an entry of $\texttt{mtf}$, which is a permutation of $[1, \sigma]$). The content of $\texttt{smallOcc}[c, o, B, V]$ is $Occ(c, o)$ for the text obtained by decompressing B starting with a move-to-front transform initialized as V. Thus, $rank_1(B^c[qt + 1, qt+t], r) = \texttt{smallOcc}[c, r, B, \texttt{mtf}[q]]$ is computed in constant time, where B is the piece of the compressed T^{bwt} starting at position $\texttt{superblockpos}[q'] + \texttt{blockpos}[q]$.

Note, however, that the table entries can be manipulated in constant time on a RAM machine only if $|B| = t' = O(\log n)$ and $|V| = O(\log n)$. The first restriction yields $t = O(\log_\sigma n)$, whereas the second becomes $\sigma = O(\log n/ \log\log n)$. The space requirement of $\texttt{smallOcc}$ is $O(\sigma t 2^{t'} \sigma! \log t)$ bits. If we choose $t = x \log_\sigma n$ for constant $0 < x < 1/3$, then $2^{t'} \le n^{3x} = o(n)$. Under this setting the overall extra space is $o(n \log \sigma)$ for $\sigma = o(\log n/ \log\log n)$.

In order to locate occurrences, Ferragina and Manzini [2000] sample text positions at regular intervals. They mark one text position out of $\log^{1+\epsilon} n$, for some $\epsilon > 0$, and collect the A values pointing to those marked positions in an array A'. To know $A[i]$, they find the smallest $r \ge 0$ such that $LF^r(i)$ is a marked position (and thus $A[LF^r(i)]$ is known), and then $A[i] = A[LF^r(i)] + r$. This way, they pay $O(n/\log^\epsilon n)$ extra space for A' and can locate the occurrences in $O(occ \log^{1+\epsilon} n)$ steps. To determine in constant time whether some $A[j]$ value is marked or not, a bit vector $\texttt{mark}_{1,n}$ tells which entries are marked. If $\texttt{mark}_j = 1$, then $A[j]$ is sampled and stored at $A'[rank_1(\texttt{mark}, i)]$. By using the techniques of Section 6.2, $\texttt{mark}$ can be stored in $O((n/\log^{1+\epsilon} n) \log n) = O(n/\log^\epsilon n)$ bits as well (as it has $n/\log^{1+\epsilon} n$ bits set). A similar approach permits displaying $T_{l,r}$ in $O(r - l + \log^{1+\epsilon} n)$ steps.

The remaining problem is that there is no easy way to know T_i^{bwt} in order to compute $LF(i) = C(T_i^{bwt}) + Occ(T_i^{bwt}, i)$. Ferragina and Manzini [2000] give an $O(\sigma)$ time solution (they assume constant σ), but a truly constant-time solution is easily obtained with a table similar to `smallOcc`: `getChar`$[o, B, V]$ returns the o-th character of the corresponding block. So each step above takes constant time.

Theorem 14 [Ferragina and Manzini 2000] *The* FM-Index *(*FMI*) offers the following space/time tradeoff.*

Space in bits	$5nH_k + o(n \log \sigma)$
Time to count	$O(m)$
Time to locate	$O(\log^{1+\epsilon} n)$
Time to display ℓ chars	$O(\ell + \log^{1+\epsilon} n)$
Conditions	$\sigma = o(\log n / \log\log n)$*;* $k \leq \log_\sigma(n/\log n) - \omega(1)$*;* $\epsilon > 0$ *is an arbitrary constant*

We note that $5nH_k$ is actually a rather pessimistic upper bound, and that the technique works with essentially any compressor for T^{bwt}. Thus the FMI obtains unbeaten counting complexity and attractive space complexity. Its real problem is the alphabet dependence, as in fact the original proposal [Ferragina and Manzini 2000] was for a constant-size alphabet. Further work on the FMI have focused on alleviating its dependence on σ.

Some more complicated techniques [Ferragina and Manzini 2000], based on using alphabet Σ^q instead of Σ, permit reducing the $O(\log^{1+\epsilon} n)$ time factor in the locating and displaying complexities to $O(\log^\epsilon n)$, yet this makes the alphabet dependence of the index even sharper.

In practice. A practical implementation of the FMI could not follow the idea of table `smallOcc` (S in their notation). Ferragina and Manzini [2001] replace `smallOcc` with a plain decompression and scanning of block B, which (according to the theoretical value of t) takes $O(\log n)$ time and raises the counting complexity to $O(m \log n)$. Some heuristics have also been used to reduce the size of the directories in practice. Also, instead of sampling the text at regular intervals, all the occurrences of some given character are sampled. This removes the need to store `mark`, as it can be deduced from the current character in T^{bwt}.

Finally, they consider alternative ways of compressing the text. The most successful one is to compress each block with a Huffman variant derived from `bzip2`, using a distinct Huffman tree per block. If we recall Theorem 4, this does not guarantee $O(nH_k)$ bits of space, but it should be close (actually, the practical implementations are pretty close to the best implementations of `bzip2`). The $O(nH_k)$ space is not guaranteed because T^{bwt} is partitioned into equal-size blocks, not according to contexts of length k. Such a partitioning will be considered in Section 9.6.

9.4 WT-FMI: An $O(nH_0)$ Size Implementation

We present now an alternative implementation of the backward search idea that is unable to reach the $O(nH_k)$ size bound, yet it is an interesting way to remove the alphabet dependence. It is called *Wavelet Tree FM-Index* (WT-FMI). The essential idea was introduced by Sadakane [2002], when wavelet trees [Grossi et al. 2003] did not yet exist. Sadakane used individual indicator arrays instead (as those proposed in the beginning of Section 6.3). The use of wavelet trees was proposed later [Ferragina et al. 2004] as a particular case of the AF-FMI (Section 9.6), and even later [Mäkinen and Navarro 2005a, 2005b] as a particular case of the RL-FMI (Section 9.5). The same idea, in the form of indicator vectors, also reappeared for the case of binary alphabets [He et al. 2005].

The idea of the WT-FMI is extremely simple, once in context (recall Theorem 3 in page 14). Just use the wavelet tree of Section 6.3 over the sequence T^{bwt}. Hence, $Occ(c, i) = rank_c(T^{bwt}, i)$ can be answered in $O(\log \sigma)$ time using the basic wavelet tree (Theorem 8), and in $O(1)$ time for $\sigma = O(\text{polylog}(n))$ using the multi-ary one (Theorem 9). The method for locating the occurrences and displaying the text is the same as for the FMI, yet this time we also find T_i^{bwt} in $O(\log \sigma)$ or $O(1)$ time using the same wavelet tree. Alg. 6 gives the pseudocode.

Algorithm WT-Occ(c, i, σ_1, σ_2, v)
(1) **if** $\sigma_1 = \sigma_2$ **then return** i;
(2) $\sigma_m = \lfloor(\sigma_1 + \sigma_2)/2\rfloor$;
(3) **if** $c \leq \sigma_m$
(4) **then return** WT-Occ(c, $rank_0(B^v, i)$, σ_1, σ_m, v_l);
(5) **else return** WT-Occ(c, $rank_1(B^v, i)$, $\sigma_m + 1$, σ_2, v_r);

Alg. 6: Computing $Occ(c, i)$ on a binary wavelet tree. It is invoked as WT-Occ$(c, i, 1, \sigma, root)$. We call B^v the bit vector at tree node v, v_l its left child, and v_r its right child.

Depending on which wavelet tree we use, different tradeoffs are obtained. We give a simplified general form that is valid for all cases. Despite its simplicity, the WT-FMI is the precursor of further research that lead to the best implementations of the backward search concept (Sections 9.5 and 9.6).

Theorem 15 *The* Wavelet Tree FM-Index *(*WT-FMI*) offers the following space/ time tradeoffs. Note that* $\log \sigma / \log \log n = O(1)$ *if* $\sigma = O(\text{polylog}(n))$*, in which case all the times are as for the* FMI.

Space in bits	$nH_0 + o(n \log \sigma)$
Time to count	$O(m(1 + \log \sigma / \log \log n))$
Time to locate	$O(\log^{1+\epsilon} n \ \log \sigma / \log \log n)$
Time to display ℓ chars	$O((\ell + \log^{1+\epsilon} n) \log \sigma / \log \log n)$
Conditions	$\sigma = o(n)$; $\epsilon > 0$ *is an arbitrary constant*

In practice: The implementation of the WT-FMI uses the binary wavelet tree, preprocessed for *rank* using the simple techniques of Section 6.1, and gives the wavelet tree the shape of the Huffman tree of the text. This way, instead of the theoretical $nH_0 + o(n \log \sigma)$ bits, we obtain $n(H_0 + 1) + o(n \log \sigma)$ bits with much simpler means [Grossi et al. 2003, 2004]. In addition, the Huffman shape gives the index $O(mH_0)$ average counting time. The worst case time is $O(m \log n)$, but this can be reduced to $O(m \log \sigma)$ without losing the $O(mH_0)$ average time. The idea is to force the Huffman tree to balance after depth $(1+x)\log\sigma$, for some constant $x>0$ [Mäkinen and Navarro 2004b].

The *Huffman FM-Index* by Grabowski et al. [2004, 2006] (Huff-FMI) obtains comparable performance and removes the alphabet dependence in another way: Sequence T^{bwt} is Huffman-compressed, and Occ is implemented using $rank$ over the binary output of Huffman. Another related approach [Ferragina 2006] uses a word-based Huffman compression (where words, not characters, are the text symbols) with byte-aligned codewords. The sequence of codewords is then indexed with an FM-Index, which is able to efficiently search for word-based queries. The space is much lower than inverted lists, which nonetheless need to store the text.

9.5 The Run-Length FM-Index of Mäkinen and Navarro (RL-FMI)

The *Run-Length FM-Index* of Mäkinen and Navarro [2004b, 2004c, 2005a, 2005c] (RL-FMI) is an improvement over the WT-FMI, which exploits the equal-letter runs of the BWT (Theorem 5) to achieve $O(nH_k(T)\log\sigma)$ bits of space. It retains the good search complexities of the FMI, but it is much more resistant to the alphabet size. Actually this was the first index achieving $O(m)$ search time for $\sigma = O(\text{polylog}(n))$ and taking simultaneously space proportional to the k-th order entropy of the text. The idea is to compute $Occ(c,i) = rank_c(T^{bwt}, i)$ using a wavelet tree built over the run-length compressed version of T^{bwt}.

In Fig. 5 we built the wavelet tree of $T^{bwt} =$ `"araadl ll$ bbaar aaaa"`. Assume that we run-length compress T^{bwt} to obtain the run heads $R =$ `"aradl l$ bar a"`. By Theorem 5, we have the limit $|R| \le nH_k(T)+\sigma^k$ for any k. Therefore, a wavelet tree built over R would require $(nH_k(T)+\sigma^k)H_0(R)+o(n\log\sigma)$ bits (Section 6.3). The only useful bound we have for the zero-order entropy of R is $H_0(R) \le \log\sigma$, thus the space bound is $nH_k(T)\log\sigma + o(n\log\sigma)$ for any $k \le \log_\sigma n - \omega(1)$.

The problem is that $rank$ over R does not give the answers we need over T^{bwt}. For example, assume we want to compute $rank_{\mathtt{a}}(T^{bwt}, 19) = 7$. We need to know that T^{bwt}_{19} lies at R_{14}. This is easily solved by defining a bitmap $\mathtt{newL}_{1,n}$ indicating the beginnings of the runs in $L = T^{bwt}$. In our case $\mathtt{newL} = 111011110111010111000$. We know that the position of T^{bwt}_{19} in R is $rank_1(\mathtt{newL}, 19) = 14$. Yet, this is not sufficient, as $rank_{\mathtt{a}}(R, 14) = 4$ just tells us that there are 4 runs of `"a"`s before and including that of T^{bwt}_{19}. What we need is to know the total length of those runs, and in which position of its run is T^{bwt}_{19} (in our case, 2nd).

For this sake, we reorder the runs in `newL` alphabetically, accordingly to the characters that form the run. Runs of the same character stay in the same relative order. We form bit array $\mathtt{sNewL}[1,n]$ with the reordered `newL`. In our case $\mathtt{sNewL} =$ 111111010100010111011. We also compute array C^R indexed by Σ, so that $C^R[c]$ tells the number of occurrences in R (runs in T^{bwt}) of characters smaller than c (thus C^R plays for R the same role C plays for L in the FMI). In our example $C^R[\texttt{"a"}] = 4$. This means that, in `sNewL`, the first $C^R[\texttt{"a"}] = 4$ runs correspond to characters smaller than `"a"`, and then come those of `"a"`, of which T^{bwt}_{19} is in the 4th because $rank_{\mathtt{a}}(R, 14) = 4$. Fig. 14 illustrates.

To compute $rank_c(T^{bwt}, i)$, we first find $i' = rank_1(\mathtt{newL}, i)$, the position of the run T^{bwt}_i belongs to in R. Thus there are $j' = rank_c(R, i')$ runs of c's in $T^{bwt}_{1,i}$. In `sNewL`, the runs corresponding to c start at $j = select_1(\mathtt{sNewL}, C^R[c]+1)$. Now there are two cases. If $R_{i'} \ne c$, then the run of T^{bwt}_i does not belong to c, and thus we must

sNewL	L	newL	R
1	a	1	a
1	r	1	r
1	a	1	a
1	a	0	
1	d	1	d
1	l	1	l
0	_	1	_
1	l	1	l
0	l	0	
1	$	1	$
0	_	1	_
0	b	1	b
0	b	0	
1	a	1	a
0	a	0	
1	r	1	r
1	_	1	_
1	a	1	a
0	a	0	
1	a	0	
1	a	0	

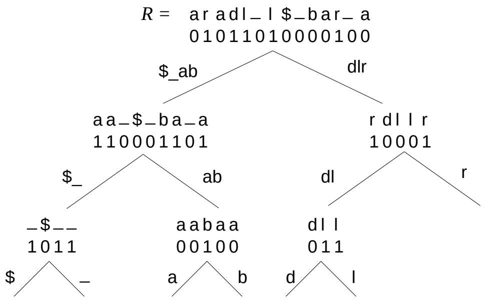

Fig. 14. The main RL-FMI structures for the text `"alabar a la alabarda$"`. The transformed text T^{bwt} is shown only for clarity. The wavelet tree on the right, built for R, stores only the bitmaps, not the texts at each node.

accumulate the full length of the first j' runs of c, $select_1(\texttt{sNewL}, C^R[c]+1+j')-j$. If, on the other hand, $R_{i'} = c$, then the run of T_i^{bwt} does belong to c, and we must count part of the last run. We are sure that the first $j'-1$ runs must be fully counted, so we have $select_1(\texttt{sNewL}, C^R[c]+j')-j$, and we must add the corresponding part of the last run, $i - select_1(\texttt{newL}, i')+1$. Alg. 7 gives the pseudocode.

Algorithm RLFM-Occ(c, i, R, newL, sNewL, C^R)
(1) $i' \leftarrow rank_1(\texttt{newL}, i)$;
(2) $j' \leftarrow rank_c(R, i')$;
(3) $j \leftarrow select_1(\texttt{sNewL}, C^R[c]+1)$;
(4) **if** $R_{i'} = c$ **then**
(5) $\quad j' \leftarrow j'-1$;
(6) $\quad ofs \leftarrow i - select_1(\texttt{newL}, i')+1$;
(7) **else** $ofs \leftarrow 0$;
(8) **return** $select_1(\texttt{sNewL}, C^R[c]+1+j')-j+ofs$;

Alg. 7: Computing $Occ(c, i)$ with the RL-FMI.

Thus the RL-FMI solves $Occ(c,i) = rank_c(T^{bwt}, i)$ in the time necessary to perform $rank_c$ over R. The rest is handled just like the WT-FMI.

Theorem 16 **[Mäkinen and Navarro 2005c]** *The* Run-Length FM-Index *(*RL-FMI*) offers the following space/time tradeoffs.*

Space in bits	$nH_k \log\sigma + 2n + o(n\log\sigma)$
Time to count	$O(m(1+\log\sigma/\log\log n))$
Time to locate	$O(\log^{1+\epsilon} n \ \log\sigma/\log\log n)$
Time to display ℓ chars	$O((\ell+\log^{1+\epsilon} n)\log\sigma/\log\log n)$
Conditions	$\sigma = o(n)$*;* $k \le \log_\sigma n - \omega(1)$*;*
	$\epsilon > 0$ *is an arbitrary constant*

In practice. The implementation of the RL-FMI, just as that of the WT-FMI (end of Section 9.4), uses binary wavelet trees with Huffman shape, with the bitmaps using the techniques of Section 6.1. This gives at most $nH_k(T)(H_0(R)+1)(1+o(1))$ space, which in the worst case is $nH_k(T)(\log\sigma+1)+o(n\log\sigma)$ bits, close to the theoretical version but much simpler to implement. In practice, $H_0(R)$ is much closer to $H_0(T)$ than to $\log\sigma$.

9.6 The Alphabet-Friendly FM-Index of Ferragina et al. (AF-FMI)

The *Alphabet-Friendly FM-Index* of Ferragina, Manzini, Mäkinen, and Navarro [2004, 2006] (AF-FMI) is another improvement over the WT-FMI of Section 9.4. The AF-FMI combines the WT-FMI technique with Theorem 4 to achieve $nH_k(T)+o(n\log\sigma)$ bits of space and the same search time of the WT-FMI.

Theorem 4 tells that, if we split T^{bwt} into substrings T^s according to its contexts s of length k, and manage to represent each resulting block T^s, of length $n_s=|T^s|$, in $n_sH_0(T^s)+f(n_s)$ bits, for any convex function f, then the sum of all bits used is $nH_k(T)+\sigma^k f(n/\sigma^k)$. In particular, we can use the binary wavelet tree of Section 9.4 for each block. It requires $n_sH_0(T^s)+O(n_s\log\log n_s/\log_\sigma n_s)$ bits (Theorem 6.3), so we need overall $nH_k(T)+O(n\log\log(n/\sigma^k)/\log_\sigma(n/\sigma^k))$ bits, for any k. If $k\le\alpha\log_\sigma n$, for any constant $0<\alpha<1$, this space is $nH_k(T)+O(n\log\log n/\log_\sigma n)$.

Assume s is the j-th nonempty context (block) in T^{bwt}. The wavelet tree of T^s allows us to solve $Occ_j(c,i)$, which is the number of occurrences of c in $T^s_{1,i}$. To answer a global $Occ(c,i)$ query, we must be able to (1) determine to which block j does i belong, so as to know which wavelet tree to query, and (2) know how many occurrences of c there are before T^s in T^{bwt}.

We store bit vector $\mathtt{newCtx}_{1,n}$ (recall Section 8.2) marking the block beginnings in T^{bwt}, so that $j=rank_1(\mathtt{newCtx},i)$. The j-th block starts at $i'=select_1(\mathtt{newCtx},j)$.

We also store a table $\mathtt{blockOcc}[j,c]$, which tells the number of occurrences of c in $T^{bwt}_{1,i'-1}$, that is, before block j (or before substring T^s). Since $\mathtt{blockOcc}[j,c]=Occ(c,i'-1)$, it follows that $Occ(c,i)=\mathtt{blockOcc}[j,c]+Occ_j(c,i-i'+1)$. Similarly, to determine T^{bwt}_i, we obtain j and i', and query the wavelet tree of the j-th block to find its $(i-i'+1)$-th character.

We saw in Section 8.2 that `newCtx` uses $O(\sigma^k\log n)$ bits of space. Table `blockOcc` requires $\sigma^{k+1}\log n$ bits. As we already have $k\le\alpha\log_\sigma n$, this extra space is $o(n)$.

Fig. 15 illustrates, for $k=1$. To determine $Occ(\texttt{"l"},8)$, we first find that $j=rank_1(\mathtt{newCtx},8)=3$ is the block number where $i=8$ belongs. The first position of block 3 is $i'=select_1(\mathtt{newCtx},3)=5$. The number of occurrences of "l" in $T^{bwt}_{1,i'-1}$, that is, before block $j=3$, is $\mathtt{blockOcc}[3,\texttt{"l"}]=0$. Inside block 3, corresponding to substring $T^s=$ "dl ll$ bb" of T^{bwt}, we need the number of "l"s in $T^s_{1,i-i'+1}=T^s_{1,4}$. This is given by the wavelet tree, which gives $Occ_3(\texttt{"l"},4)=2$. Thus the answer is $0+2=2$.

Yet, Ferragina et al. [2004] go further. Instead of choosing a fixed k value in advance, they use a method by Ferragina, Giancarlo, Manzini, and Sciortino [2005] that, given a space overhead function $f(n_s)$ on top of $n_sH_0(T^s)$, finds the partition of T^{bwt} that optimizes the final space complexity. Using blocks of fixed context length k is just one of the possibilities considered in the optimization, so the resulting partition is below $nH_k(T)+\sigma^k f(n/\sigma^k)$ *simultaneously for all* k (and it is possibly better than using any fixed k). That is, although we have made the anal-

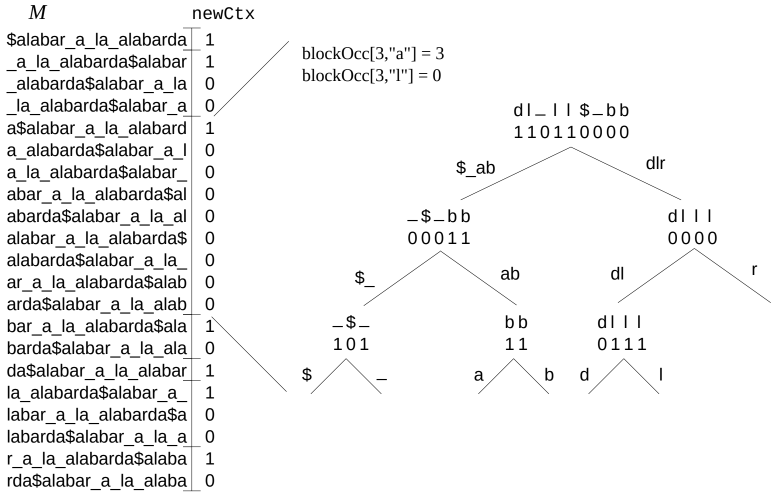

Fig. 15. The main AF-FMI structures for the text `"alabar a la alabarda$"` considering contexts of length $k = 1$. Matrix M of the BWT is shown only for clarity. We show only one of the wavelet trees, corresponding to context `"a"`, and only a couple of its Occ_j values. Note that this wavelet tree corresponds to a substring of that in Fig. 5.

ysis assuming a given k, the construction does not have to choose any k but it reaches the space bound for any k of our choice. Thus this index also achieves the independence of k mentioned at the end of Section 8.2, yet it obtains at the same time the minimum space nH_k, using significantly simpler means.

The locating of occurrences and displaying of text is handled just as in Section 9.4.

Theorem 17 [Ferragina et al. 2006] *The* Alphabet-Friendly FM-Index *(*AF-FMI*) offers the following space/time tradeoffs.*

Space in bits	$nH_k + o(n \log \sigma)$
Time to count	$O(m(1 + \log \sigma / \log \log n))$
Time to locate	$O(\log^{1+\epsilon} n \log \sigma / \log \log n)$
Time to display ℓ chars	$O((\ell + \log^{1+\epsilon} n) \log \sigma / \log \log n)$
Conditions	$\sigma = o(n)$*;* $\epsilon > 0$ *is an arbitrary constant;*
	$k \leq \alpha \log_\sigma n$*, for some constant* $0 < \alpha < 1$

10. LEMPEL-ZIV BASED INDEXES

Up to now we have considered different ways of compressing suffix arrays. While this is clearly the most popular trend on compressed indexing, it is worthwhile to know that there exist alternative approaches to self-indexing, based on Lempel-Ziv compression. In particular, one of those requires $O(m + occ)$ time to locate the occ occurrences of P in T. This has not been achieved with other indexes.

10.1 Lempel-Ziv Compression

In the seventies, Lempel and Ziv [1976] presented a new approach to data compression. It was not based on text statistics, but rather on identifying repeated text substrings and replacing repetitions by pointers to their former occurrences in T. Lempel-Ziv methods produce a *parsing* (or partitioning) of the text into *phrases*.

Definition 16 *The* LZ76 parsing *[Lempel and Ziv 1976] of text $T_{1,n}$ is a sequence $Z[1,n']$ of phrases such that $T = Z[1]\,Z[2]\ldots Z[n']$, built as follows. Assume we have already processed $T_{1,i-1}$ producing sequence $Z[1,p-1]$. Then, we find the longest prefix $T_{i,i'-1}$ of $T_{i,n}$ which occurs in $T_{1,i-1}$.*[6] *If $i' > i$ then $Z[p] = T_{i,i'-1}$ and we continue with $T_{i',n}$. Otherwise T_i has not appeared before and $Z[p] = T_i$, continuing with $T_{i+1,n}$. The process finishes when we obtain $Z[n'] =$ "\$".*

When phrase $T_{i,i'-1}$ is found within $T_{1,i-1}$, say at $T_{j,j+(i'-1-i)}$, we say that $T_{j,j+(i'-1-i)}$ is the *source* of phrase $T_{i,i'-1}$. The output of a LZ76-based compressor is essentially the sequence of pairs $(j, i'-i)$ (new characters in Σ that appear are exceptions in this encoding). An important property of LZ76 is that every phrase has already appeared before, unless it is a new character of Σ.

We will also use a Lempel-Ziv parsing called LZ78, where each phrase is formed by an already known phrase concatenated with a new character at the end.

Definition 17 *The* LZ78 parsing *[Ziv and Lempel 1978] of text $T_{1,n}$ is a sequence $Z[1,n']$ of phrases such that $T = Z[1]\,Z[2]\ldots Z[n']$, built as follows. The first phrase is $Z[1] = \varepsilon$. Assume we have already processed $T_{1,i-1}$ producing a sequence $Z[1,p-1]$ of $p-1$ phrases. Then, we find the longest prefix of $T_{i,n}$ which is equal to some $Z[p']$, $1 \le p' < p$. Thus $T_{i,n} = Z[p']\,c\,T_{i',n}$, $c \in \Sigma$. We define $Z[p] = Z[p']\,c$ and continue with $T_{i',n}$. The process finishes when we get $c =$ "\$".*

The output of an LZ78 compressor is essentially the sequence of pairs (p', c) found at each step p of the algorithm. Note two facts: (1) all the phrases in an LZ78 parsing are different from each other; (2) the prefixes of a phrase are phrases.

Fig. 16 shows the LZ78 parsing of our example text (among other structures we review soon). For example, $Z[9] =$ `"lab"`. See also Fig. 17, which illustrates a structure that is conceptually important for LZ78: The *Lempel-Ziv trie* is the trie storing the set of strings Z. This trie has exactly n' nodes (one per string in Z). If $Z[p] = Z[p']\,c$, then node p is a child of p' by edge labeled c.

An important property of both Lempel-Ziv parsings is the following.

Lemma 7 *Let n' be the number of phrases produced by LZ76 or LZ78 parsing of text $T_{1,n}$ over an alphabet of size σ. Then $n' = O(n/\log_\sigma n)$.*

To show that the lemma holds for LZ78 it suffices to notice that all the phrases are different, and therefore we can have only σ^ℓ phrases of length ℓ. If we try to maximize n' by using first the phrases of length 1, then length 2, and so on, we use $O(n/\log_\sigma n)$ phrases to cover n characters. In LZ76 we can have repeated phrases, but no phrase $Z[p]$ can repeat more than σ times before all the longer phrases $Z[p]\,c$ are already known. From then on, $Z[p]$ cannot appear alone again.

[6] The original definition [Lempel and Ziv 1976] actually permits the former occurrence of $T_{i,i'-1}$ to extend beyond position $i-1$, but we ignore this feature here.

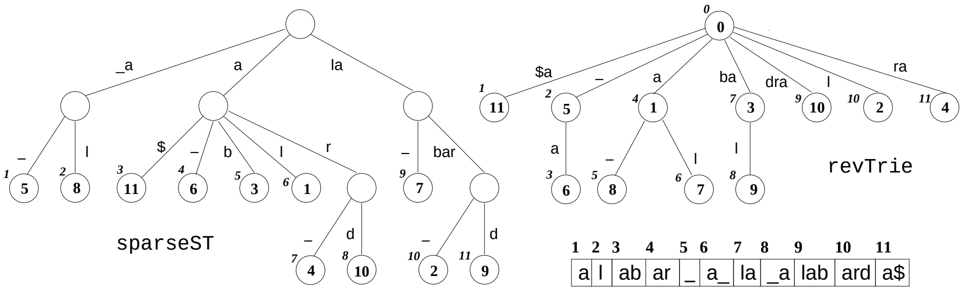

Fig. 16. On the bottom-right, the LZ78 parsing of the text `"alabar a la alabarda$"`. The rest are some structures of the KU-LZI: `sparseST` and `revTrie`. We use phrase identifiers as node labels. The smaller numbers in italics outside the nodes are the lexicographic ranks of the corresponding strings.

Furthermore, the size of the Lempel-Ziv compressed text (slowly) converges to the entropy of the source [Cover and Thomas 1991]. Of more direct relevance to us is that n' is related to the empirical entropy $H_k(T)$ [Kosaraju and Manzini 1999; Ferragina and Manzini 2005].

Lemma 8 *Let n' be the number of phrases produced by LZ76 or LZ78 parsing of text $T_{1,n}$. Then $n' \log n = nH_k(T) + O((k+1)n' \log \sigma)$. As $n' \le n/\log_\sigma n$, this is $nH_k(T) + o(n \log \sigma)$ for $k = o(\log_\sigma n)$.*

Lemma 8 implies that a tree with n' nodes can be stored, even using pointers, in $O(nH_k)$ bits of space. We can even store a constant number of integers per node.

The pioneer work in Lempel-Ziv based indexes, and also the first compressed index we know of (albeit not a self-index), is due to Kärkkäinen and Ukkonen [1996a]. It derives from their earlier work on sparse suffix trees [Kärkkäinen and Ukkonen 1996b], which we briefly review before entering into Lempel-Ziv based methods. The sparse suffix tree, which is in turn the first succinct index we know of, is a suffix tree indexing every h-th text position. It easily finds the aligned occurrences in $O(m)$ time. The others can start up to $h-1$ positions after a sampled position. Thus we search for all the patterns of the form $\Sigma^i P$, $0 \le i < h$. Overall this requires $O(\sigma^{h-1}(h+m) + occ)$ time. By choosing $h = 1 + \epsilon \log_\sigma n$ we get $O(n^\epsilon(m + \log_\sigma n) + occ)$ search time and $O((n \log n)/h) = O(n \log \sigma)$ bits.

10.2 The LZ-Index of Kärkkäinen and Ukkonen (KU-LZI)

The *LZ-Index* of Kärkkäinen and Ukkonen [1996a] (KU-LZI) uses a suffix tree that indexes only the beginnings of phrases in a LZ76-like parsing of T. Although they only prove (using Lemma 7) that their index is succinct, taking $O(n \log \sigma)$ bits of space, Lemma 8 shows that it actually requires $O(nH_k)$ bits of space (plus text). We present the results in their definitive form [Kärkkäinen 1999]. As the exact parsing is not essential in their method, we use the LZ78 parsing to exemplify it.

The LZ76 parsing has the property that each new phrase has already appeared in T, or it is a new character in Σ. Thus, the first occurrence of any pattern P cannot be completely inside a phrase, otherwise it would have appeared before (the exception is $m = 1$, which is easy to handle and we disregard here). This is also true in LZ78 parsing. They divide occurrences among *primary* (spanning two or more phrases) and *secondary* (completely inside a phrase). Secondary occurrences are repetitions of other primary or secondary occurrences. Assume there are occ_p primary and occ_s secondary occurrences, so that $occ = occ_p + occ_s$.

Fig. 16 illustrates two main structures of the KU-LZI. The suffix tree indexing only phrase beginnings is `sparseST`; and `revTrie` is a trie storing the reverse phrases (unary paths are compressed in `revTrie`, recall Definition 5 in page 5).

Primary occurrences are found as follows. For some $1 \leq i < m$, $P_{1,i}$ is the suffix of a phrase and $P_{i+1,m}$ starts at the next phrase. To ensure that each occurrence is reported only once, we require that $P_{1,i}$ is completely included in a phrase, so that the partitioning $(P_{1,i}, P_{i+1,m})$ is unique. The (phrase-aligned) occurrences of $P_{i+1,m}$ are found using `sparseST`. The occurrences of $P_{1,i}$ within (and at the end of) a phrase are found by searching `revTrie` for $P_i P_{i-1} \ldots P_1$. For example, $P =$`"labar"` appears in phrase 2 split as (`"l"`,`"abar"`) and in phrase 9 as (`"lab"`,`"ar"`).

Each of those two searches yields a *lexicographical range* in $[1, n']$. Tree `sparseST` yields the range $[l_2, r_2]$ of the phrase-aligned suffixes that start with $P_{i+1,m}$, whereas `revTrie` gives the range $[l_1, r_1]$ of phrases that finish with $P_{1,i}$. Consider now the p-th phrase. Assume $Z[p]$ reversed is ranked x_p-th among all reversed phrases, and that the suffix starting at phrase $p + 1$ is ranked y_p-th among all phrase-aligned suffixes. Then we wish to report a primary occurrence (with P_i aligned at the end of $Z[p]$) iff $(x_p, y_p) \in [l_1, r_1] \times [l_2, r_2]$. We use a two-dimensional range search structure `range` (Section 6.4) to store the n' points (x_p, y_p) and search for the range $[l_1, r_1] \times [l_2, r_2]$. For example, for $P =$`"labar"` and $i = 3$, `revTrie` finds range $[l_1, r_1] = [8, 8]$ for `"bal"` (as only the 8th reverse phrase starts with `"bal"`, see Fig. 16) and `sparseST` finds range $[l_2, r_2] = [7, 8]$ for `"ar"` (as the 7th and 8th suffixes starting phrases start with `"ar"`). Then the search for $[8, 8] \times [7, 8]$ in `range` finds point $(7, 8)$ corresponding to $p = 9$.

Secondary occurrences are obtained by tracking the source of each phrase $Z[p]$. Given a primary occurrence $T_{j,j+m-1}$, we wish to find all phrases p whose source contains $[j, j + m - 1]$. Those phrases contain secondary occurrences $T_{j',j'+m-1}$, which are again tracked for new copies. With some slight changes to the LZ76 parsing, it can be ensured that no source contains another and thus source intervals can be linearly ordered (by their start or end positions, both orders coincide). An array `source`$[i]$ of the phrases sorted by their source interval position in T, plus a bit array `newSrc`$_{1,n}$ indicating which text positions start phrase sources, permits finding each phrase that copies area $[j, j+m-1]$ in constant time. We want those sources that start not after j and finish not before $j + m - 1$: `source`$[rank_1($`newSrc`$, j)]$ is the last phrase in `source` whose source starts not after j. We traverse `source` backwards from there until the source intervals finish before $j + m - 1$. Each step in this traversal yields a new secondary occurrence. Alg. 8 gives the pseudocode.

The index space is $O(n' \log n) = O(nH_k(T))$ for `sparseST` and `revTrie`, as both have $O(n')$ nodes. Among the range search data structures considered by Kärkkäinen [1999], we take those requiring $O(\frac{1}{\epsilon} n' \log n')$ bits of space (the one we reviewed in Section 6.4 corresponds to $\epsilon = 1$). Array `source` also needs the same space, and bit array `newSrc` requires $O(n' \log n)$ bits using the techniques of Section 6.2. Thus the overall space is $O(\frac{1}{\epsilon} nH_k(T))$ bits, in addition to the text.

We note that this index carries out counting and locating simultaneously. The

Algorithm LZ-Report(j, m, `sources`, `newSrc`)
output j;
$i \leftarrow rank_1(\texttt{newSrc}, j)$;
while $i > 0 \ \wedge\ \texttt{sources}[i].end \geq j + m - 1$ **do**
    LZ-Report $(\texttt{sources}[i].target + (j - \texttt{sources}[i].start), m, \texttt{sources}, \texttt{newSrc})$;
    $i \leftarrow i - 1$;

Alg. 8: Reporting occurrence $[j, j + m - 1]$ and all the secondary occurrences that copy it. We assume that entries `sources`$[i]$ have fields *start* and *end* (where the source of the phrase starts and ends), and *target* (start position where the phrase is copied to).

$m-1$ searches in `revTrie` and `sparseST` add up $O(m^2)$ time. Primary occurrences are found with `range`, which takes $O(\log n)$ time per search plus $O(\frac{1}{\epsilon}\log^{\epsilon} n)$ time per occurrence. Secondary occurrences are found in constant time.

Theorem 18 [Kärkkäinen and Ukkonen 1996a] *The* LZ-Index *of Kärkkäinen and Ukkonen (*KU-LZI*) offers the following space/time tradeoff.*

Space in bits	$O(\frac{1}{\epsilon}nH_k) + o(n\log\sigma) + n\log\sigma$
Time to count	$O(m^2 + m\log n + \frac{1}{\epsilon}occ\log^{\epsilon} n)$
Time to locate	*free after counting*
Time to display ℓ chars	$O(\ell)$ *(text is available)*
Conditions	$k = o(\log_\sigma n)$*;* $0 < \epsilon < 1$

The first term of the counting complexity can be made $O(m^2/\log_\sigma n)$ by letting the tries move by $O(\log_\sigma n)$ characters in one step, yet this raises the space requirement to $O(n\log\sigma)$ unless we use much more recent methods [Grossi et al. 2003]. By using range search data structures that appeared later [Alstrup et al. 2000], the index would require $O(nH_k\log^{\gamma} n)$ bits and count in $O(m^2 + m\log\log n + occ)$ time. Also, a variant of this index [Kärkkäinen and Sutinen 1998] achieves $O(m)$ counting time and $O(occ)$ locating time, but only for short patterns ($m < \log_\sigma n$).

The *LZ-Index* of Navarro [2002, 2004] (Nav-LZI) is an evolution on the KU-LZI. It improves the space complexity by converting the index into a self-index. The counting time complexity is not competitive, but the locating and displaying times are good. The Nav-LZI unbeaten in this aspect in terms of empirical performance.

10.3 The LZ-Index of Ferragina and Manzini (FM-LZI)

The *LZ-Index* of Ferragina and Manzini [2005] (FM-LZI) is the only existing self-index taking $O(m)$ counting time and constant time to locate each occurrence. It is based on the LZ78 parsing of T (Definition 17) and requires $O(nH_k\log^{\gamma} n)$ bits of space for any constant $\gamma > 0$.

Let us define $T^{\#}$ as the text T where we have inserted special characters "#" after each phrase (so $|T^{\#}| = n + n'$). For our example text T = `"alabar a la alabarda$"` we have $T^{\#}$ = `"a#l#ab#ar# #a #la# a#lab#ard#a$#"`. We also define T^R as text $T^{\#}$ read backwards, T^R = `"#$a#dra#bal#a #al# a# #ra#ba#l#a"`. Let A be the suffix array of T and A^R that of T^R. Finally, let $P^R = \#P_m P_{m-1}\ldots P_1$.

The FM-LZI consists of four components: (1) the FMI of text T; (2) the FMI of text T^R; (3) `lzTrie`, the Lempel-Ziv trie of T; (4) `range`, a structure similar to that of the KU-LZI. The first three structures require $O(nH_k(T))$ bits of space, yet this time range will dominate the space complexity. As the FMI of T is enough for counting in $O(m)$ time, we will focus in locating the *occ* occurrences in $O(m + occ)$ time. Occurrences of P are divided into primary and secondary as in Section 10.2 (they are called "external" and "internal" by Ferragina and Manzini [2005]).

Let us first consider secondary occurrences. Since every prefix of a phrase is also a phrase, every secondary occurrence which is not at the end of its phrase p occurs also in the phrase p' referenced by p (that is, the parent of p in lzTrie). Fig. 17 depicts the lzTrie. For example, pattern $P =$"a" occurs in phrase 10. Since it does not occur at the end of $Z[10] =$"ard", it must also occur in its parent 4, $Z[4] =$"ar" and in turn in its parent 1, $Z[1] =$"a". Let us call a trie node p a *pioneer* for P if P is a suffix of $Z[p]$. In Fig. 17 the pioneer nodes for $P =$"a" are 1, 7, and 8. Then, all secondary occurrences correspond to lzTrie subtrees rooted at pioneer nodes. Thus, to find those occurrences, we obtain the pioneer nodes and traverse all their subtrees reporting all the text positions found (with the appropriate offsets).

The pioneer nodes are found with the FMI of T^R. We search for P^R, which corresponds to occurrences of $P\#$ in $T^\#$, that is, occurrences of P that are phrase suffixes. For example, if we search for $P^R =$"#a" in T^R we will find occurrences at positions 12, 15, and 31 of T^R (see Fig. 17). This corresponds to the occurrences of "a#" in $T^\#$, at positions 20, 17, and 1, respectively. Aligned to the (contiguous) area of A^R corresponding to suffixes that start with "#", we store a vector lzNode of pointers to the corresponding lzTrie nodes. As the range for P^R is always contained in the area covered by lzNode, this permits finding the pioneer nodes of the results of the search. Thus the occ_s secondary occurrences are reported in $O(m + occ_s)$ time. As lzNode has n' entries, it occupies $nH_k(T) + o(n \log \sigma)$ bits.

Let us now consider the primary occurrences. The same idea of Section 10.2, of searching for $P_{1,i}$ at the end of a phrase and $P_{i+1,n}$ from the next phrase, is applied. Yet, the search proceeds differently, and the FMI is shown to be a very fortunate choice for this problem. We first search for P using the FMI of T. This single

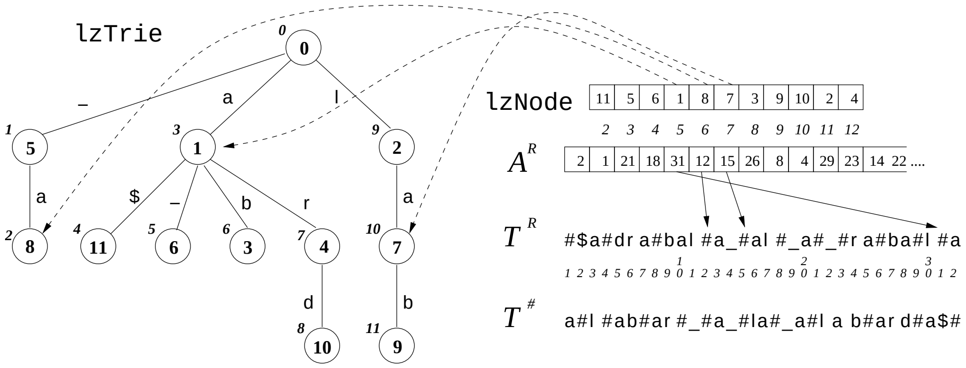

Fig. 17. Parts of the FM-LZI. We show $T^\#$, T^R, and A^R (none of which is explicitly represented), as well as vector lzNode and lzTrie. Only the part of A^R pointing to suffixes starting with "#" is shown in detail. This is the part lzNode is aligned to. For legibility, lzNode shows phrase numbers instead of pointers to lzTrie. The result of the search for "#a" is illustrated with the actual pointers from A^R to T^R and from lzNode to lzTrie.

search gives us all the ranges $[sp_{i+1}, ep_{i+1}]$ in A corresponding to the occurrences of $P_{i+1,m}$, $1 \le i < m$ (recall Section 9.1). We now search for P^R in the FMI of T^R. After we have each range $[sp'_i, ep'_i]$ corresponding to the occurrences of $P_i P_{i-1} \ldots P_1$ in A^R, we add character "#", obtaining the range $[sp_i^R, ep_i^R]$ of the occurrences of $\# P_i P_{i-1} \ldots P_1$ in A^R, $1 \le i < m$. This corresponds to occurrences of $P_{1,i}\#$ in $T^\#$. Note that the search in T^R ensures that $P_{1,i}$ is completely contained in a phrase (and is at the end of it), while the search in T permits $P_{i+1,m}$ to span as many phrases as necessary. All this process takes $O(m)$ time.

Consider searching for $P =$`"bar"`. There are $m-1 = 2$ possible partitions for P, (`"b"`,`"ar"`) and (`"ba"`,`"r"`). Those appear in $T^\#$ as `"b#ar"` and `"ba#r"`. Using the FMI of T we get $[sp_3, ep_3] = [20, 21]$ (A range for `"r"`), and $[sp_2, ep_2] = [12, 13]$ (A range for `"ar"`), see Figs. 3 and 8. Using the FMI of T^R we get $[sp_3^R, ep_3^R] = [8, 9]$ (A^R range for `"#b"`, corresponding to `"b#"` in $T^\#$), and we get that $[sp_2^R, ep_2^R]$ (A^R range for `"#ab"`, corresponding to `"ba#"` in $T^\#$) is empty, see Fig. 17. Thus, we know that it is possible to find primary occurrences corresponding to partition (`"b"`,`"ar"`). The first part corresponds to $A^R[8, 9]$ and the second to $A[12, 13]$. Looking at Figs. 3 and 17, we see that $A^R[8] = 26$ in T^R continues with $A[12] = 5$ in T and $A^R[9] = 8$ in T^R continues with $A[13] = 17$ in T. The mapping of positions is as follows: position r in T, belonging to the p-th phrase, maps to $rev(r) = (n + n' + 1) - (r + p - 1)$ in T^R.

Structure `range`, storing n' points in $[1, n] \times [1, n]$, is used to find those matching pairs. Let $j = A^{-1}[r]$ be the position in A pointing to $T_{r,n}$, where r starts the p-th phrase in T. Similarly, let $j^R = (A^R)^{-1}[rev(r) + 1]$ be the position in A^R pointing to $T^R_{rev(r)+1,n}$ (which represents $T_{1,r-1}$). We store pairs (j^R, j) in `range`. A range search for $[sp_i^R, ep_i^R] \times [sp_{i+1}, ep_{i+1}]$ retrieves all those phrase positions r such that $P_{1,i}$ is a suffix of the phrase preceding position r and $P_{i+1,m}$ follows in $T_{r,n}$. Thus we report text positions $r - i + 1$, where each occurrence P starts. In our example, two points we would store are $(8, 12)$ and $(9, 13)$, corresponding to $r = 5$ and $r = 17$ in T. These will be retrieved by range query $[8, 9] \times [12, 13]$.

Ferragina and Manzini [2005] use for `range` the structure of Alstrup et al. [2000] (see Section 6.4) that can store n' points in $[1, n'] \times [1, n']$ using $O(n' \log^{1+\gamma} n')$ bits for any $\gamma > 0$, so that they answer a query with res results in time $O(\log\log n' + res)$. In our case, we must query the structure once per each partition $1 \le i < m$, so we pay overall $O(m \log\log n + occ_p)$. Note that our points are actually in $[1, n] \times [1, n]$. Those can be mapped to $[1, n'] \times [1, n']$ using *rank* and *select* on bitmaps of length n with n' bits set. Using the techniques of Section 6.2, those bitmaps require $O(n' \log n) = O(nH_k(T))$ bits. Note that the space of the structure, $O(n' \log^{1+\gamma} n')$ bits, is $O(nH_k(T) \log^\gamma n) + o(n \log\sigma \log^\gamma n)$ if $k = o(\log_\sigma n)$.

The $O(m \log\log n)$ time can be improved as follows. Basically, instead of storing only the positions r that start a phrase in `range`, we add all positions $[r - \log\log n + 1, r]$. Now each cut $(P_{1,i}, P_{i+1,m})$ would be found $\log\log n$ times, not once. Thus we can search only for those i that are multiples of $\log\log n$. As we perform only $m/\log\log n$ queries, the overall time is $O(m + occ_p)$. Although now we store $n' \log\log n$ points in `range`, the space complexity stays the same. We omit some technical details to handle borders between phrases.

For patterns shorter than $\log\log n$ we must use a different approach. Those patterns are so short that we can precompute all their primary occurrences with a four-Russians technique. There are at most $\sigma^{\log\log n} = (\log n)^{\log \sigma}$ different short patterns, each requiring a pointer of $\log n$ bits to its occurrence list, and the total number of primary occurrences for all short patterns is at most $n'(\log\log n)^2$ (as they must start at most $\log\log n$ positions before a phrase border, and finish at most $\log\log n$ positions after it), each requiring $\log n$ bits as well. The overall space for short patterns is $o(n \log\sigma \log^\gamma n)$ if $\sigma = o(n^{1/\log\log n})$. For example, this is valid whenever $\sigma = O(n^\beta)$ for any $0 < \beta < 1$.

Text contexts are displayed using the same FMI. By using the AF-FMI rather than the original FMI, we obtain the following result, where counting time is $O(m)$ for $\sigma = O(\text{polylog}(n))$.

Theorem 19 [Ferragina and Manzini 2005] *The* LZ-Index *of Ferragina and Manzini (*FM-LZI*) offers the following space/time tradeoffs.*

Space in bits	$O(nH_k \log^\gamma n) + o(n \log\sigma \log^\gamma n)$
Time to count	$O(m(1 + \log\sigma/\log\log n))$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell + \log^{1+\epsilon} n)$
Condition	$\gamma > 0$ *is any constant*
Space in bits	$O(nH_k \log\log n) + o(n \log\sigma \log\log n)$
Time to count	$O(m(1 + \log\sigma/\log\log n))$
Time to locate	$O(\log\log n)$
Time to display ℓ chars	$O(\ell + \log^{1+\epsilon} n)$
Conditions for all	$\sigma = o(n^{1/\log\log n})$*;* $k = o(\log_\sigma n)$*;* $0 < \epsilon < 1$ *is any constant*

The second version uses other results by Alstrup et al. [2000], which search in time $O((\log\log n)^2 + res \log\log n)$ using $O(n' \log n' \log\log n')$ bits. We retain our counting time by indexing $(\log\log n)^2$ positions per phrase instead of $\log\log n$.

The two-dimensional range search idea has inspired other solutions to achieve constant time per occurrence on compressed suffix arrays [He et al. 2005], yet those work only for sufficiently large m.

11. DISCUSSION

We have presented the main ideas of several compressed indexes as intuitively as possible, yet with an accuracy enough to understand the space/time tradeoffs they achieve. In many cases, these depend on several parameters in a complex way, which makes a fair comparison difficult. Table 1 provides a rough summary.

It is interesting at this point to discuss the most important common points in these approaches. Common points within the CSA family and within the FMI family are pretty obvious, namely they are basically different implementations of functions Ψ and Occ, respectively. There is also an obvious relation among these two families, as Ψ and LF are the inverse of each other (Lemma 4).

What is more subtle is the relation between the different ways to achieve $O(nH_k)$ space. Let us exclude the Lempel-Ziv based methods, as they are totally different. For this discussion, the table of Fig. 13 is particularly enlightening. Each number

Table 1. Simplified space and time complexities for compressed full-text indexes. For space complexities, we present only the main term related to entropies and seek to minimize space. For time complexities, we present bounds that hold on some representative inputs, assuming for example that the alphabet is small enough. We refer to the theorems given earlier for accurate statements and boundary conditions. Recall that γ and ϵ are small positive constants.

Self	*Index*	entropy term	time to count + locate	Theorem	page
N	GV-CSA	nH_0	$O(m \log^2 n + occ \log n)$	10	33
Y	SAD-CSA	nH_0	$O(m \log n + occ \log n)$	11	35
Y	GGV-CSA	nH_k	$O(m \log \sigma + \log^3 n + occ \log^2 n)$	13	40
Y	FMI	$5nH_k$	$O(m + occ \log^{1+\epsilon} n)$	14	44
Y	WT-FMI	nH_0	$O(m + occ \log^{1+\epsilon} n)$	15	45
Y	RL-FMI	$nH_k \log \sigma$	$O(m + occ \log^{1+\epsilon} n)$	16	47
Y	AF-FMI	nH_k	$O(m + occ \log^{1+\epsilon} n)$	17	49
N	KU-LZI	$O(nH_k)$	$O(m^2 + (m + occ) \log n)$	18	53
Y	FM-LZI	$O(nH_k \log^\gamma n)$	$O(m + occ)$	19	56

$j \in [1, n]$ can be classified according to two parameters: the list (or table column) $T_{A[j]-1}$ and the context (or table row) $T_{A[j],A[j]+k-1}$ where j belongs. Within a given cell, numbers j (sharing $T_{A[j]-1,A[j]+k-1}$), are classified according to $T_{A[j]+k,n}$.

As A is sorted by $T_{A[j],n}$ (a refinement of the k-th context order), all the j values in each table row form a contiguous subinterval of $[1, n]$, which advances with the row number. How the j values within each row (corresponding to contexts $T_{A[j],A[j]+k-1}$) distribute across columns, depends on $T_{A[j]-1}$, the characters preceding the occurrences of the context in T.

Instead of row-then-column, consider now a column-then-row order. Now j values are collected in the order given by $T_{A[j]-1,n}$, or renaming $j = \Psi(i)$ (as Ψ is a permutation), $\Psi(i)$ values are collected in the order given by $T_{A[\Psi(i)]-1,n} = T_{A[i],n}$. This order is of course $i = 1$, 2, and so on, thus we are in fact reading array Ψ. Numbers j are increasing inside each column because they are ordered by $T_{A[j],n}$.

The SAD-CSA structure stores Ψ in order, that is, it stores the table in column-wise order, and then row-wise inside each column. Being σ increasing lists, this leads to $O(nH_0)$ space. The GGV-CSA, instead, stores Ψ in row-wise order. For this sake, it needs to record how the values inside each row distribute across columns. According to Theorem 4, it is sufficient to store that distribution information in space close to its zero-order entropy, to achieve nH_k overall space. The GGV-CSA uses one wavelet tree per row to represent the column each element belongs to.

In a widely different view, the AF-FMI structure stores T^{bwt} context-wise (that is, row-wise in the table). For each context, it stores the characters of T^{bwt}, which are precisely $T^{bwt}_j = T_{A[j]-1}$, that is, the column identifiers of the positions j lying within each context (row). The AF-FMI uses the same wavelet tree to represent basically the same data within the same zero-order entropy space. Thus both structures are using essentially the same concept to achieve nH_k space.

The differences are due to other factors. While the GGV-CSA structure still adheres to the idea of abstract optimization of A, so that it must provide access to $A[j]$ and use the normal binary search on A, the FMI family uses a completely different form of searching, which directly builds on T^{bwt}.

The final practical twist of the GGV-CSA is the discovery that γ- or δ-encoding of consecutive j values within each cell of the table yields $O(nH_k)$ space, independently of whether one uses row-wise or column-wise order (as there are not too many jumps across table cells if k is small enough). This permits a much simpler implementation of the structure, which turns out to be close to the Sad-CSA, initially believed to require $O(nH_0)$ space.

The other indexes use a widely different mechanism to achieve $O(nH_k)$ space. They rely on compressing the runs that appear in Ψ, or similarly those in T^{bwt}. The former (Mak-CSA and MN-CCSA, not covered in this survey) achieve $O(nH_k \log n)$ space by emulating the binary search on A through Ψ, whereas the latter achieve $O(nH_k \log \sigma)$ space by emulating backward search strategy.

Table 2 classifies the approaches that reach H_k-related space according to their approach. In one dimension, we have those based on local entropy (Theorem 4) versus those based on run lengths (Theorem 5). In the other dimension, we have those based on Ψ or on *Occ*. We have included Sad-CSA as having size nH_k according to our findings in this paper (yet, remind it needs $O(n \log\log \sigma)$ extra space). We classify the FMI as using run lengths because this is the key property ensuring its $O(nH_k)$ size, although it also uses some local entropy optimization. Recall that we left aside Lempel-Ziv methods in this discussion and in the table.

Table 2. Classifying the suffix array indexes with size related to H_k. Names are followed by a pair indicating (main space term , simplified counting time) complexities.

	local entropy	run lengths
using Ψ	Sad-CSA (nH_k, $m \log n$)	Mak-CSA ($2nH_k \log n$, $m \log n$)
	GGV-CSA (nH_k, $m \log \sigma$)	MN-CCSA ($nH_k \log n$, $m \log n$)
using *Occ*	AF-FMI (nH_k, m)	FMI ($5nH_k$, m)
		RL-FMI ($nH_k \log \sigma$, m)

12. CONCLUSIONS

We have given a unified look at the state of the art in compressed full-text indexing. We focused on the essential ideas relating text compressibility and regularities on indexes built on it, and uncovered fundamental relations between seemingly disparate approaches. Those efforts have led to a rich family of results, whose most important consequence is a surprising fact of text compressibility:

Fact. *Instead of compressing a text into a representation that does not reveal anything from the original text unless decompressed, one can obtain an almost equally space-efficient representation that in addition provides fast searching on the text.*

In other words, the indexes we have reviewed take space close to what can be obtained by the best possible compressors, both in theory and in practice. In theory, the leading term in the space complexities of the best indexes is $nH_k(T)$, which is a lower-bound estimate for many text compression techniques. For substring searches, the same best indexes are practically optimal, obtaining $O(m)$ counting query time. This remarkable discovery is without any doubt one of the most important achievements ever obtained in text compression and text indexing.

However, there are some open questions to be answered. A first one is whether one can obtain $nH_k(T)$ space and $O(m)$ query time on any alphabet size, and in general which is the lower bound relating these parameters. Recently, Gagie [2006] shows that, as soon as $\sigma^{k+1/c-3\varepsilon} = \Omega(n)$, it is not possible to represent T using $cnH_k(T) + \varepsilon n \log \sigma$ bits of space. This implies, for example, that in Theorem 17 (and others alike) one could not loosen the condition on k to $k \le \log_\sigma n - O(1)$. Other bounds apply to the sublinear space complexities, which might be bigger than the entropy-related part. Recent lower bounds [Miltersen 2005; Golynski 2006] on *rank* and *select* dictionaries show that only limited progress can be expected in this direction. On the other hand, k-th order entropy might not be the best compressibility measure, as for example it can be beaten with run-length compression.

Another open challenge is to obtain better output sensitivity in reporting queries within little space. For this goal, there are some results achieving $O(occ+o(n))$ time for large enough m [Grossi and Vitter 2000, 2006], $O(occ)$ time for large enough m [He et al. 2005], and even $O(occ)$ time without any restriction on m, for not very large alphabets, using $O(nH_k(T)\log^\gamma n)$ bits of space [Ferragina and Manzini 2005]. The technique by He et al. [2005] is general and can be plugged into any of the indexes discussed before, by adding some sublinear-size dictionaries.

In this survey we have focused on the most basic problem, namely exact search in main memory. There are many further challenges, with regard to more complex searching, index construction and updating, secondary memory, and so on. A brief list of other relevant problems beyond the scope of this survey follows.

Secondary memory. Although their small space requirements might permit compressed indexes to fit in main memory, there will always be cases where they have to operate on disk. There is not much work yet on this important issue. One of the most attractive full-text indexes for secondary memory is the String B-tree [Ferragina and Grossi 1999], among others [Ko and Aluru 2006]. These are not, however, succinct structures. Some proposals for succinct and compressed structures in this scenario exist [Clark and Munro 1996; Mäkinen et al. 2004]. A good survey on full-text indexes in secondary memory is due to Kärkkäinen and Rao [2003]. See also [Aluru 2005, Chapter 35].

Construction. Compressed indexes are usually derived from an uncompressed one. Although it is usually simple to build a classical index and then derive its compressed version, there might not be enough space to build the uncompressed index first. Secondary memory might be available, but many classical indexes are costly to build in secondary memory. Therefore, an important problem is how to build compressed indexes without building their uncompressed versions first. Several papers have recently appeared on the problem of building the Sad-CSA in little space [Lam et al. 2002; Hon et al. 2003; Hon et al. 2003; Na 2005], as well as the Nav-LZI [Arroyuelo and Navarro 2005] and the WT-FMI [Mäkinen and Navarro 2006]. There is also some recent work on efficient construction of (plain) suffix arrays (see [Puglisi et al. 2006] for a good survey). With respect to construction of (plain) indexes in secondary memory, there is a good experimental comparison for suffix arrays [Crauser and Ferragina 2002], as well as some work on suffix trees [Farach et al. 2000; Clifford 2005]. For further details on the topic, see [Aluru 2005, Chapters 5 and 35].

Dynamism. Most indexes considered are static, in the sense that they have to be rebuilt from scratch upon text changes. This is currently a problem even on uncompressed full-text indexes, and not much has been done. Yet, there is some recent work on compressed indexes [Ferragina and Manzini 2000; Hon et al. 2004; Chan et al. 2004; Mäkinen and Navarro 2006].

Extended functionality. We have considered only exact string matching in this survey, yet classical full-text indexes permit much more sophisticated search tasks, such as approximate pattern matching, regular expression matching, pattern matching with gaps, motif discovery, and so on [Apostolico 1985; Gusfield 1997]. There has been a considerable amount of work on extending compressed suffix arrays functionalities to those of suffix trees [Grossi and Vitter 2000; Munro et al. 2001; Sadakane 2002; Sadakane 2003; Grossi et al. 2004; Kim and Park 2005; Grossi and Vitter 2006]. The idea in general is to permit the simulation of suffix tree traversals using a compressed representation of them, such as a compressed suffix array plus a parentheses representation of the suffix tree shape [Munro and Raman 1997]. In addition, there has been some work on approximate string matching over compressed suffix arrays [Huynh et al. 2006; Lam et al. 2005; Chan et al. 2006]. Finally, it is also interesting to mention that the idea of backward searching has been used to search plain suffix arrays in $O(m \log \sigma)$ time [Sim et al. 2003].

Technology transfer. An extremely important aspect is to make the transfer from theory to technology. Already several implementations exist for most indexes surveyed in this article, showing the proof-of-concept and the practicality of the ideas. It is matter of more people becoming aware of the intriguing opportunities provided by these new techniques, for a successful technology transfer to take place. To facilitate the chance for smooth transfer from prototype implementations to real use, a repository of standardized library implementations has been made available at the *PizzaChili* site (see page 4). Articles about the FM-Index have appeared in popular journals such as *DrDobbs Journal* (December 2003) and in *CT Magazine* (January 2005). Also, the bioinformatics community is becoming aware of the techniques [Healy et al. 2003]. Finally, several recent papers on the topic can be found in practical venues, such as *Efficient and Experimental Algorithms (WEA)*.

Overall, we believe that self-indexing is among the most exciting research areas in text compression and text indexing, which in a few years has obtained striking results and has a long way ahead, rich in challenges and possibly new surprises.

Acknowledgements

We thank Paolo Ferragina, Kunihiko Sadakane, and the anonymous referees for their invaluable help to improve this survey. Their hard work is greatly appreciated.

REFERENCES

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2*, 1, 53–86.

Alstrup, S., Brodal, G., and Rauhe, T. 2000. New data structures for orthogonal range searching. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)* (2000), pp. 198–207.

Aluru, S. 2005. *Handbook of Computational Molecular Biology.* CRC Press.

ANDERSSON, A. AND NILSSON, S. 1995. Efficient implementation of suffix trees. *Software Practice and Experience 25*, 2, 129–141.

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

ARLAZAROV, V., DINIC, E., KONROD, M., AND FARADZEV, I. 1975. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady 11*, 1209–1210.

ARROYUELO, D. AND NAVARRO, G. 2005. Space-efficient construction of LZ-index. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS v. 3827 (2005), pp. 1143–1152.

BAEZA-YATES, R. AND RIBEIRO, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

BELL, T., CLEARY, J., AND WITTEN, I. 1990. *Text compression*. Prentice Hall.

BENTLEY, J., SLEATOR, D., TARJAN, R., AND WEI, V. 1986. A locally adaptive compression scheme. *Communications of the ACM 29*, 4, 320–330.

BLUMER, A., BLUMER, J., HAUSSLER, D., MCCONNELL, R., AND EHRENFEUCHT, A. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM 34*, 3, 578–595.

BURROWS, M. AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

CHAN, H.-L., HON, W.-K., AND LAM, T.-W. 2004. Compressed index for a dynamic collection of texts. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3109 (2004), pp. 445–456.

CHAN, H.-L., LAM, T.-W., SUNG, W.-K., TAM, S.-L., AND WONG, S.-S. 2006. A linear size index for approximate pattern matching. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009 (2006), pp. 49–59.

CHAZELLE, B. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing 17*, 3, 427–462.

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

CLARK, D. AND MUNRO, I. 1996. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1996), pp. 383–391.

CLIFFORD, R. 2005. Distributed suffix trees. *Journal of Algorithms 3*, 2–4, 176–197.

COLUSSI, L. AND DE COL, A. 1996. A time and space efficient data structure for string searching on large texts. *Information Processing Letters 58*, 5, 217–222.

COVER, T. AND THOMAS, J. 1991. *Elements of Information Theory*. Wiley.

CRAUSER, A. AND FERRAGINA, P. 2002. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32*, 1, 1–35.

CROCHEMORE, M. AND VÉRIN, R. 1997. Direct construction of compact directed acyclic word graphs. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 1264 (1997), pp. 116–129.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry — Algorithms and Applications*. Springer-Verlag.

ELIAS, P. 1975. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory 21*, 2, 194–203.

FARACH, M. 1997. Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Symposium on Foundations of Computer Science (FOCS)* (1997), pp. 137–143.

FARACH, M., FERRAGINA, P., AND MUTHUKRISHNAN, S. 2000. On the sorting complexity of suffix tree construction. *Journal of the ACM 47*, 6, 987–1011.

FERRAGINA, P. 2006. String algorithms and data structures. In *Algorithms for Massive Data Sets*, Lecture Notes in Computer Science, Tutorial Book (2006). Springer-Verlag. To appear.

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. and Grossi, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM 46*, 2, 236–280.

Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)* (2000), pp. 390–398.

Ferragina, P. and Manzini, G. 2001. An experimental study of an opportunistic index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2001), pp. 269–278.

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

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2004. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 3246 (2004), pp. 150–160.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2006. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*. To appear. Also as TR 2004-05, Technische Fakultät, Universität Bielefeld, Germany, December 2004.

Fredkin, E. 1960. Trie memory. *Communications of the ACM 3*, 490–500.

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

Geary, R., Rahman, N., Raman, R., and Raman, V. 2004. A simple optimal representation for balanced parentheses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3109 (2004), pp. 159–172.

Giegerich, R., Kurtz, S., and Stoye, J. 2003. Efficient implementation of lazy suffix trees. *Software Practice and Experience 33*, 11, 1035–1049.

Golynski, A. 2006. Optimal lower bounds for rank and select indexes. In *Proc. 33th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS v. 4051 (2006), pp. 370–381.

Golynski, A., Munro, I., and Rao, S. 2006. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (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.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. 2005. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)* (Greece, 2005), pp. 27–38. CTI Press and Ellinika Grammata.

Grabowski, S., Mäkinen, V., and Navarro, G. 2004. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 3246 (2004), pp. 210–211.

Grabowski, S., Navarro, G., Przywarski, R., Salinger, A., and Mäkinen, V. 2006. A simple alphabet-independent FM-index. *International Journal of Foundations of Computer Science (IJFCS)*. To appear. See also Technical Report TR/DCC-2004-4, Department of Computer Science, University of Chile, July 2004.

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

Grossi, R., Gupta, A., and Vitter, J. 2004. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2004), pp. 636–645.

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

GROSSI, R. AND VITTER, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35*, 2, 378–407.

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

HE, M., MUNRO, I., AND RAO, S. 2005. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2005), pp. 23–32.

HEALY, J., THOMAS, E. E., SCHWARTZ, J. T., AND WIGLER, M. 2003. Annotating large genomes with exact word matches. *Genome Research 13*, 2306–2315.

HON, W.-K., LAM, T.-W., SADAKANE, K., AND SUNG, W.-K. 2003. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)* (2003), pp. 240–249.

HON, W.-K., LAM, T.-W., SADAKANE, K., SUNG, W.-K., AND YU, S.-M. 2004. Compressed index for dynamic text. In *Proc. 14th IEEE Data Compression Conference (DCC)* (2004), pp. 102–111.

HON, W.-K., SADAKANE, K., AND SUNG, W.-K. 2003. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS)* (2003), pp. 251–260.

HUYNH, T., HON, W.-K., LAM, T.-W., AND SUNG, W.-K. 2006. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science 352*, 1–3, 240–249.

IRVING, R. 1995. Suffix binary search trees. Technical Report TR-1995-7 (April), Computer Science Department, University of Glasgow, UK.

ITOH, H. AND TANAKA, H. 1999. An efficient method for in-memory construction of suffix arrays. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE)* (1999), pp. 81–88. IEEE CS Press.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)* (1989), pp. 549–554.

KÄRKKÄINEN, J. 1995. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 937 (1995), pp. 191–204.

KÄRKKÄINEN, J. 1999. *Repetition-Based Text Indexing*. Ph. D. thesis, Department of Computer Science, University of Helsinki, Finland.

KÄRKKÄINEN, J. AND RAO, S. 2003. *Algorithms for Memory Hierarchies*, Chapter 7: Full-text indexes in external memory, pp. 149–170. LNCS v. 2625. Springer.

KÄRKKÄINEN, J. AND SANDERS, P. 2003. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS v. 2719 (2003), pp. 943–955.

KÄRKKÄINEN, J. AND SUTINEN, E. 1998. Lempel-Ziv index for q-grams. *Algorithmica 21*, 1, 137–154.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996a. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)* (1996), pp. 141–155. Carleton University Press.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996b. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS v. 1090 (1996), pp. 219–230.

KIM, D. AND PARK, H. 2005. A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3537 (2005), pp. 33–44.

KIM, D., SIM, J., PARK, H., AND PARK, K. 2005. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3*, 2–4, 126–142.

KIM, D.-K., NA, J.-C., KIM, J.-E., AND PARK, K. 2005. Efficient implementation of rank and select functions for succinct representation. In *Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, LNCS v. 3503 (2005), pp. 315–327.

Knuth, D. 1973. *The Art of Computer Programming*, Volume 3: Sorting and Searching. Addison-Wesley.

Ko, P. and Aluru, S. 2005. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3*, 2–4, 143–156.

Ko, P. and Aluru, S. 2006. Obtaining provably good performance from suffix trees in secondary storage. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009 (2006), pp. 72–83.

Kosaraju, R. and Manzini, G. 1999. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing 29*, 3, 893–911.

Kurtz, S. 1998. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Universität Bielefeld, Germany.

Lam, T.-W., Sadakane, K., Sung, W.-K., and Yiu, S.-M. 2002. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics (COCOON)* (2002), pp. 401–410.

Lam, T.-W., Sung, W.-K., and Wong, S.-S. 2005. Improved approximate string matching using compressed suffix data structures. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS v. 3827 (2005), pp. 339–348.

Larsson, N. and Sadakane, K. 1999. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Department of Computer Science, Lund University, Sweden.

Lempel, A. and Ziv, J. 1976. On the complexity of finite sequences. *IEEE Transactions on Information Theory 22*, 1, 75–81.

Mäkinen, V. 2000. Compact suffix array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 1848 (2000), pp. 305–319.

Mäkinen, V. 2003. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae 56*, 1–2, 191–210.

Mäkinen, V. and Navarro, G. 2004a. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3109 (2004), pp. 420–433.

Mäkinen, V. and Navarro, G. 2004b. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report C-2004-20 (April), University of Helsinki, Finland.

Mäkinen, V. and Navarro, G. 2004c. Run-length FM-index. In *Proc. DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"* (Aug. 2004), pp. 17–19.

Mäkinen, V. and Navarro, G. 2005a. Succinct suffix arrays based on run-length encoding. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3537 (2005), pp. 45–56.

Mäkinen, V. and Navarro, G. 2005b. Succinct suffix arrays based on run-length encoding. Technical Report TR/DCC-2005-4 (March), Department of Computer Science, University of Chile, Chile.

Mäkinen, V. and Navarro, G. 2005c. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12*, 1, 40–66.

Mäkinen, V. and Navarro, G. 2006. Dynamic entropy-compressed sequences and full-text indexes. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009 (2006), pp. 307–318.

Mäkinen, V., Navarro, G., and Sadakane, K. 2004. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS v. 3341 (2004), pp. 681–692.

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

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

Manzini, G. and Ferragina, P. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40*, 1, 33–50.

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

Miltersen, P. 2005. Lower bounds on the size of selection and rank indexes. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2005), pp. 11–12.

Morrison, D. 1968. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15*, 4, 514–534.

Munro, I. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180 (1996), pp. 37–42.

Munro, I. and Raman, V. 1997. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Symposium on Foundations of Computer Science (FOCS)* (1997), pp. 118–126.

Munro, I., Raman, V., and Rao, S. 2001. Space efficient suffix trees. *Journal of Algorithms 39*, 2, 205–222.

Na, J.-C. 2005. Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3537 (2005), pp. 57–67.

Navarro, G. 2002. Indexing text using the ziv-lempel trie. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 2476 (2002), pp. 325–336.

Navarro, G. 2004. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms 2*, 1, 87–114.

Navarro, G., Moura, E., Neubert, M., Ziviani, N., and Baeza-Yates, R. 2000. Adding compression to block addressing inverted indexes. *Information Retrieval 3*, 1, 49–77.

Pagh, R. 1999. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP)* (1999), pp. 595–604.

Puglisi, S., Smyth, W., and Turpin, A. 2006. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*. To appear.

Raman, R. 1996. Priority queues: small, monotone and trans-dichotomous. In *Proc. 4th European Symposium on Algorithms (ESA)*, LNCS v. 1136 (1996), pp. 121–137.

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

Rao, S. 2002. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters 82*, 6, 307–311.

Sadakane, K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Symposium on Algorithms and Computation (ISAAC)*, LNCS v. 1969 (2000), pp. 410–421.

Sadakane, K. 2002. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2002), pp. 225–232.

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

Sadakane, K. and Okanohara, D. 2006. Practical entropy-compressed rank/select dictionary. Manuscript. `http://arxiv.org/abs/cs.DS/0610001`.

Schürmann, K. and Stoye, J. 2005. An incomplex algorithm for fast suffix array construction. In *Proc. 7th Workshop on Algorithm Engineering and Experiments and 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)* (2005), pp. 77–85. SIAM Press.

Sedgewick, R. and Flajolet, P. 1996. *An Introduction to the Analysis of Algorithms*. Addison-Wesley.

Sim, J., Kim, D., Park, H., and Park, K. 2003. Linear-time search in suffix arrays. In *Proc. 14th Australasian Workshop on Combinatorial Algorithms (AWOCA)* (2003), pp. 139–146.

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

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

Witten, I., Moffat, A., and Bell, T. 1999. *Managing Gigabytes* (2nd ed.). Morgan Kaufmann Publishers.

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory 24*, 5, 530–536.

Ziviani, N., Moura, E., Navarro, G., and Baeza-Yates, R. 2000. Compression: A key for next-generation text retrieval systems. *IEEE Computer 33*, 11, 37–44.

APPENDIX

A. TABLE OF SYMBOLS

symbol	explanation	section	page
Σ	alphabet	2	4
σ	alphabet size, $\sigma = \|\Sigma\|$	2	4
c	arbitrary symbol from Σ	2	4
S	arbitrary string — sequence of symbols from Σ	2	4
T	text — (long) string to be indexed and queried	2	4
n	length of T, $n = \|T\|$	2	4
P	pattern — (short) string to search for	2	4
m	length of P, $m = \|P\|$	2	4
$\$$	end marker, smaller than other symbols, $\$ \in \Sigma$	2	4
occ	number of occurrences of P in T	2	4
A	suffix array of T	3.3	8
sp, ep	start/end position of occurrences of P in A	3.3	8
$H_0(S)$	zeroth order entropy of S, $H_0 = H_0(T)$	5	18
$H_k(S)$	k-th order entropy of S, $H_k = H_k(T)$	5	18
k	length of a text context	5	17
s	arbitrary text context in T, $\|s\| = k$	5	17
T^s	characters preceding context s in T	5	18
n_s	length of T^s, $n_s = \|T^s\|$ (equals the number of occurrences of s in T)	5	18
n_c	number of occurrences of symbol c in T	5	18
$\Psi(i)$	permutation built on A, $A[\Psi(i)] = A[i] + 1$	5	20
T^{bwt}	Burrows-Wheeler transform of T, $T_i^{bwt} = T_{A[i]-1}$	5.3	21
M	matrix of cyclic shifts of T in lexicographic order	5.3	21
F	first column of M	5.3	21
L	last column of M, $L = T^{bwt}$	5.3	21
LF	mapping from L to F, $LF(i) = C(L_i) + Occ(L_i, i)$	5.3	22
$C(c)$	number of symbols smaller than c in T	5.3	22
$Occ(c, i)$	number of occurrences of symbol c up to position i (usually within string T^{bwt})	5.3	22
B	arbitrary bit vector	6	24
t	block size when cutting a bit vector into blocks	6	25
κ	number of bits set in a bit vector	6	27
Z	Lempel-Ziv parsing of T, $T = Z[1]\,Z[2] \ldots Z[n']$	10	49
n'	number of phrases in Z, $n' = \|Z\|$	10	49