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, and 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 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 up to date, focusing on the essential aspects on how they exploit the text compressibility and how they solve efficiently various search problems. We aim at giving the theoretical 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. The volume of new material published in the Web in the period 1999–2003 exceeds what has been produced during the whole previous history of Mankind. A large part of these data is formed by *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. 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, boosting the potential of text as 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. On natural language text, for example, information retrieval and natural language processing techniques aim at discovering the meaning of text passages and their relevance to the information need of a human. The main tools are statistical analysis of occurrences and variants of words. Another prominent example is computational biology, which works on DNA, protein, or gene sequences to extract information on evolutionary history, biochemical function, chemical structure, and so on. The main tool used is sequence comparison to determine similarities according to diverse biological models.

There is a basic search task, however, that underlies all the applications that aim at extracting information from text. *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 or retrieving sequences most similar to a sample. Significant developments in basic string matching have a wide impact on most applications. In the rest of the paper we focus on basic string matching (also called "text searching" in some scenarios).

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 wide availability of massive storage. The real problem, however, is efficient access. In the last two decades, the CPU speeds have been doubling every 18 months, while the disk access times have stayed basically unchanged. Computer caches are many times faster than their 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. Moreover, most existing indexes are not designed to work in secondary memory, so using them from disk is extremely inefficient. 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 index.

In the analysis above, we are excluding a very well-known particular case. If the text consists of *natural language* and we stick to word and phrase queries, then *inverted indexes* require only 20%– 100% of extra space on top of the text [Baeza-Yates and Ribeiro 1999], which is very low by the above standards. 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, it is important to notice the limitations of this approach. First, the keyword "natural language" excludes several very important Human languages. It refers only to languages where words can be syntactically separated and follow some statistical laws such as Heaps' (which governs the number of distincts words) and Zipf-Mandelbrot (which governs their frequency distribution) [Baeza-Yates and Ribeiro 1999]. This includes 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; as well as agglutinating languages, where particles are glued to form long "words", yet one wishes to search for the particles (such as Finnish and German). Second, natural language excludes many 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.

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 Ziv-Lempel 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 in some widely used indexing data structures. Several others succinct and self-indexes appeared almost simultaneously [Mäkinen 2000; Grossi and Vitter 2000; Sadakane 2000]. The exciting 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 2000; Ferragina and Manzini 2005; Grossi et al. 2003; Ferragina et al. 2006]. Yet, several of these developments are still in a theoretical stage, and significant algorithm engineering is necessary to implement them. However, some promising steps have already been taken in making the compressed indexes more usable in real-world scenarios. For example, there are studies on construction time and space [Hon et al. 2003], management of secondary storage [Mäkinen et al. 2004], searching for more complex patterns [Huynh et al. 2004], updating upon text changes [Hon et al. 2004], etc. 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 explain the relationship between the compressibility of a text and the regularities that show up in its indexing structures. Those regularities can be exploited to compress the indexes. Then, we cover the most relevant self-indexes that have been proposed, focusing on the conceptual foundations underlying them, and explaining how they exploit the regularities of the indexes to compress them and at the same time search them efficiently.

We do not give experimental results in this paper. Doing this carefully and thoroughly would require many more pages. Some comparisons can be found, for example, by Mäkinen and Navarro [2005a]. All the implementations we refer to across the survey can be found from our Web pages. A repository of texts and standardized implementations of succinct full-text indexes is available at mirrors `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`.

2. NOTATION AND BASIC CONCEPTS

We introduce the notation we use in the rest of the paper. 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, for technical convenience, 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 sequence S is 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 sequence S' at the end of S. It is also possible to concatenate string S and character c, as in cS or Sc. This is the same as if c were a string of length 1.

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 we 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 would be multipied $O(\log n)$. In other words, the dependency on the machine model is not severe.

3. BASIC TEXT INDEXES

In this section we cover the most popular full-text indexes, which are in particular those that are turned into compressed indexes later in this paper. We are interested in general indexes that work for any kind of text. As explained in the Introduction, this excludes in particular the popular inverted indexes. Yet, we will consider inverted indexes in Section 4.5 as an example of a compressed index for natural language text.

Given the focus of this paper, we are also 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. 1999].

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 corresponds to a distinct prefix in the set. The root node corresponds to 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.

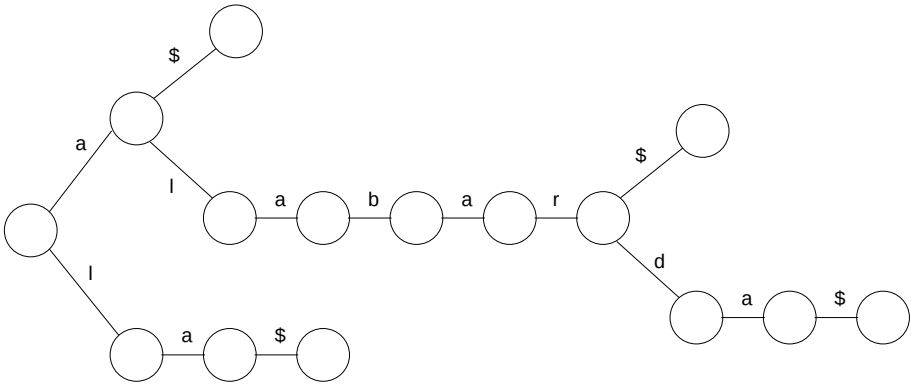

Fig. 1. A trie for the set {"alabar", "a", "la", "alabarda"}. In general trie nodes may have arity up to σ.

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 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 permits $O(1)$ search time $O(1)$ space factor at each node, 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 in the set prefixed by S. We can 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).

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. In this case, it is possible that we arrive at a leaf during the search for P, without having read the complete suffix of T. This means that P has at most one occurrence in T. We have to follow the pointer and compare the rest of P with the rest of the suffix of T to determine whether there is one occurrence or none.

With or without the trick of pruning nodes, the trie of a text $T_{1,n}$ requires space and construction time $O(n^2)$ in the worst case. Albeit this worst case is unlikely, and on average the trie requires $O(n)$ space [Sedgewick and Flajolet 1996], there exist equally powerful structures that guarantee this 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.

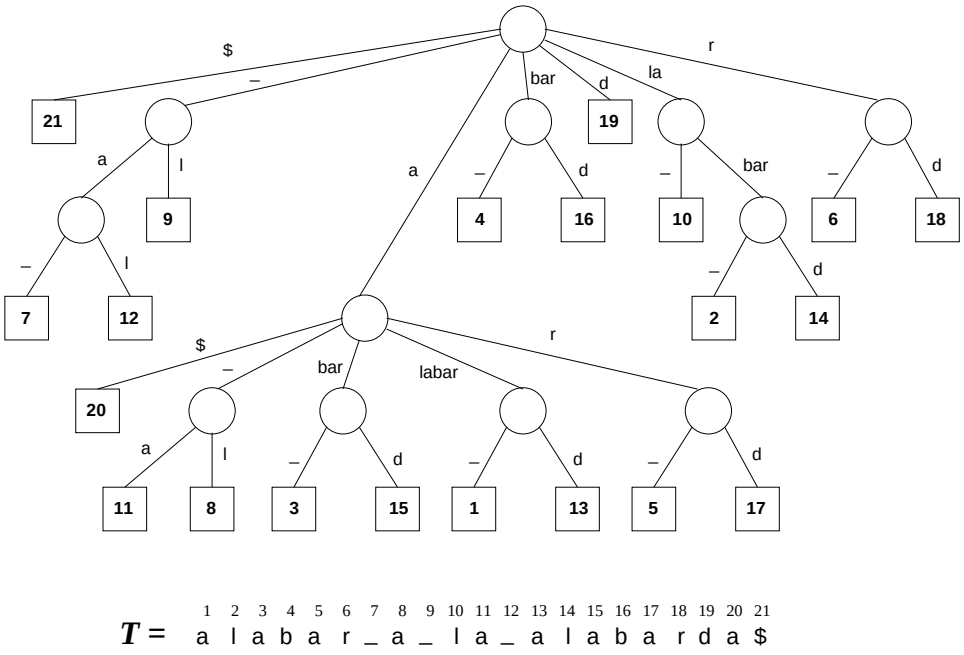

Fig. 2. The suffix tree of the text `"alabar a la alabarda$"`. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters `"a"`–`"z"`.

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 an internal 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 text 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 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]. Also, in practice perfect hashing is replaced either by binary searching on the array of existing children (so m is replaced by $m \log \sigma$ in the time complexities), or by a direct indexing into a full array of all the possible children (so n is replaced by σn in the space complexities).

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 $1 \leq i < n$, 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 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. 2003; Ko and Aluru 2003; 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].

Fig. 3 shows our example suffix array. Note that each subtree of the suffix tree corresponds to the subinterval in the suffix array encompassing all its leaves (in the figure we have highlighted the interval corresponding to the suffix tree node that descends from the root by character `"a"`).

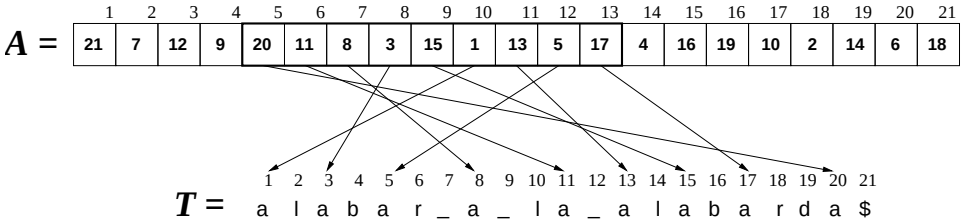

Fig. 3. The suffix array of the text `"alabar a la alabarda$"`. We have shown explicitly where the suffixes starting with `"a"` point to.

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. 2002]). A locating query requires additional $O(occ)$ time to report the occ occurrences. Fig. 4 gives the pseudocode.

```
Algorithm SASearch(P_{1,m},A[1,n],T_{1,n})
(1)  sp ← 1; st ← n;
(2)  while sp < st do
(3)      s ← ⌊(sp + st)/2⌋;
(4)      if P > T_{A[s],A[s]+m−1} then sp ← s + 1 else st ← s;
(5)  ep ← sp − 1; et ← n;
(6)  while ep < et do
(7)      e ← ⌈(ep + et)/2⌉;
(8)      if P = T_{A[e],A[e]+m−1} then ep ← e else et ← e − 1;
(9)  return (sp, ep);
```

Fig. 4. Algorithm to search for P in suffix array A over 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 $A[i]$, for $sp \leq i \leq ep$.

All the space/time tradeoffs offered by the different succinct 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 INDEXES

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

We explain two elementary concepts that have a significant role in compressed full-text indexes. Suprisingly, these two concepts can be plugged to traditional full-text indexes making impact beyond compression. In fact, no knowledge of compression techniques is required to understand the power of these methods. These two concepts 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 the familiar *inverted index* can be easily turned into a compressed self-index for natural language texts. Then we will 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 Fig. 4. Ferragina and Manzini [2000] propose a completely reverse way of guiding the search: The pattern is searched for from its last character on to its first character. Fig. 5 illustrates how this backward search proceeds.

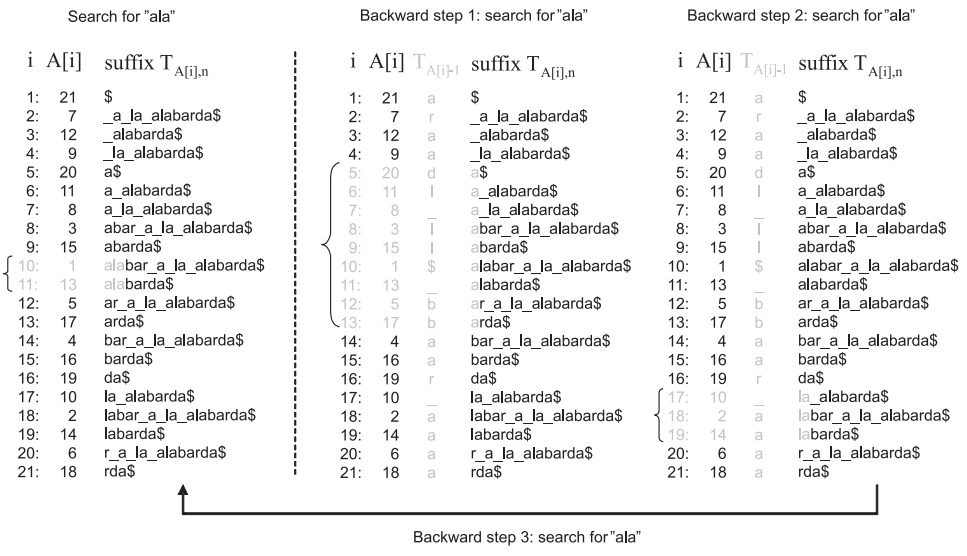

Fig. 5. Backward search for pattern `"ala"` on the suffix array of the text `"alabar a la alabarda$"`.

Fig. 5 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 $[5, 13]$ in suffix array A where the suffixes start with `"a"`. This is trivial: Everything one needs is an array C indexed by the characters, such that C[`"a"`] 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[`"b"`] $- 1$.

To understand step 2 in Fig. 5, consider 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 $T_{A[5],21}$, $T_{A[6],21}$, ..., $T_{A[13],21}$ are the *only ones* starting with `"a"`, we know that suffixes $T_{A[5]-1,21}$, $T_{A[6]-1,21}$, ..., $T_{A[13]-1,21}$ are the *only candidates* to start with `"la"`: We just need to check which of those candidates actually start with `"l"`.

The only crux is to efficiently find the range corresponding to suffixes starting with `"la"` after knowing the range corresponding to `"a"`. This is the novel point needing more formalism, but to get an idea, consider all the concatenated suffixes `"l"` $T_{A[i],n}$ in the descending row order they appear in (that is, find the rows where $T_{A[i]-1}$ = `"l"`). Now find the rows for the corresponding suffixes $T_{A[i]-1,n}$ = `"l"` $T_{A[i],n}$. Row 6 becomes row 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 and after the rows $[5, 13]$ in column $T_{A[i]-1}$. Both counts are zero, and hence the complete range $[17, 19]$ of suffixes starting with `"l"` 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 after row 19 in column $T_{A[i]-1}$. The counts are $\underline{5}$ and $\underline{2}$, respectively. This means that the range corresponding to suffixes starting with `"ala"` is $[C$[`"a"`]$+\underline{5}, C$[`"b"`]$- 1 - \underline{2}] = [10, 11]$.

We have now discovered that backward search can be implemented by means of table C (to map each character to the lexicographically smallest suffix starting with it) and some queries on the column $T_{A[i]-1}$. Let us denote column $T_{A[i]-1}$ by string $L_{1,n}$ (for reasons to make clear later). When formalized, one notices that the *single* query needed on L is counting how many times a given character appears *up to* some given position. Let us denote this query $Occ(c, i)$ on character c and position i. For example, in Fig. 5 we have L =`"raadl_ll$_bbaar_aaaa"` and Occ(`"a"`$, 16) = \underline{5}$ (and Occ(`"a"`$, 19) = 7 = 9 - \underline{2}$, where 9 is the total number of times `"a"` appears in L).

Another way of deriving the backward search algorithm (see Section 9) is to notice that function $LF(i) = C[L[i]] + Occ(L[i], i)$ maps suffix $T_{A[i]...n}$ to suffix $T_{A[i]-1...n}$ (this permits decoding the text backwards). For example, $C[L[14]] + Occ(L[14], 14) = C$[`"a"`$] + Occ$(`"a"`$, 14) = 4 + 4 = 8$ (from suffix `"bar a la alabarda$"` to suffix `"abar a la alabarda$"`). Instead of mapping one suffix at the time, the backward search maps a range of suffixes (those whose prefix

matches the current pattern suffix $P_{i...m}$) to their predecessors having the required first character P_{i-1} (by induction, the suffixes in the new range have the common prefix $P_{i-1...m}$).

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 $rank_1(B, i) = Occ(c, i)$. 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 bis 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 afore-studied $Occ(c, i)$ queries is the *wavelet tree* Grossi, Gupta, and Vitter [2003]. The idea is to simulate each $Occ(c, i)$ query by $\log \sigma$ $rank_1$-queries on binary sequences, as illustrated in Fig. 6.

The structure of the wavelet tree is a balanced search tree where each symbol from the alphabet corresponds to a leaf in the tree (see Fig. 6). The root of the wavelet tree holds a bit-vector marking with 1 those character positions that belong to the right subtree. Characters on those marked positions are concatenated to form the sequence corresponding to the right child of the root. Symmetrically, characters on the unmarked positions form the sequence corresponding to the left child of the root. The same marking of character positions and concatenation of marked/unmarked positions is repeated recursively until the leaves. Only the bit-vectors marking the positions are stored, and they are preprocessed for $rank_1$-queries.

Fig. 6 shows how $Occ($`"a"`$, 15)$ is computed, where $L =$`"raadl_ll$_bbaar_aaaa"`. As we know that `"a"` belongs to the first half of the sorted alphabet, it receives mark 0 in the root bit-vector, and consequently its occurrences go to the left child.

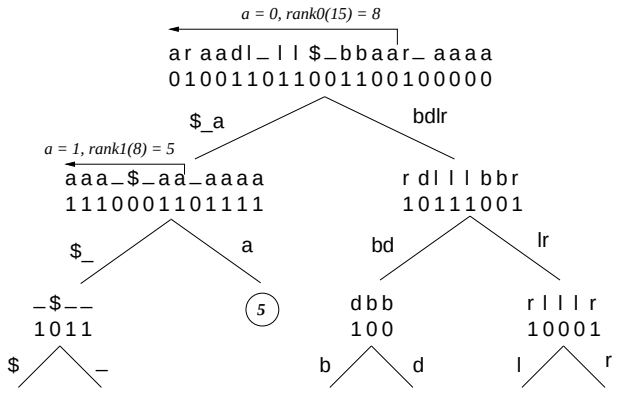

Fig. 6. A binary wavelet tree for the string $L =$ `"raadl_ll$_bbaar_aaaa"`, illustrating the solution of query $Occ($`"a"`$, 15)$. Only the bit vectors are stored, the texts are shown for clarity.

Thus, we compute $rank_0(B, 15) = 15 - rank_1(B, 15) = 15 - 7 = 8$ to find out which is its corresponding character position in the concatenated 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 $rank_1(B', 8) = 5$ to find out the corresponding position in the right child of the current node. As that child is a leaf, it would contain a sequence formed by just `"a"`s, thus we already have our answer $Occ($`"a"`$, 15) = 5$. (Readers familiar with computational geometry might notice that the process is quite identical to *fractional cascading* [de Berg et al. 2000, Chapter 5].) In practice, knowing which bit mark corresponds to a character can be made very easy: One can use the bits of the integer representation of the character within Σ, from most to least significant.

Some care in the implementation (see Section 6.3) leads to $n \log \sigma + o(n \log \sigma)$ bits representation of the wavelet tree 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.

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 counting queries can actually be supported without using suffix arrays at all! Essentially, the wavelet tree for a sequence derived from the original text suffices. For locating the occurrence positions or displaying text context, the combination of suffix array and the original text is still necessary. The main

technique to cope without the suffix array nor the text is to sample both at regular intervals, so that the space needed to store the samples is sublinear. 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 uses the backward step mechanism $LF(i)$ to find the suffix array entry pointing to text position $A[i]-1$, $A[i]-2$, ..., 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, following its stored link to obtain the corresponding suffix array position i such that $A[i] = r$, and then using function LF successively from i to print out one character per step from r to l (for example, the wavelet tree reveals the character at a given position in $\log \sigma$ steps). Complete descriptions of these procedures are given in the next sections (the details slightly varying from index to index).

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

4.4 Forward Searching: Compressed Suffix Arrays

Another line of studies on self-indexes [Grossi and Vitter 2000; Sadakane 2000] builds on the *inverse* of $LF(i) = C[[L[i]] + Occ(L[i], i)$ function studied above (see Section 5.3 for details). The inverse function, denoted Ψ, maps suffix $T_{A[i]...n}$ to suffix $T_{A[i]+1...n}$, and thus enables scanning the text in forward direction, from left to right. Continuing our example, we computed $LF(14) = C[L[14]]+Occ(L[14], 14) = 8$, and thus the inverse is $\Psi(8) = 14$. In general, the mapping has the simple definition $\Psi(i) = j$ such that $A[j] = A[i]+1$. Fig. 7 illustrates.

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

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. 7 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 was used in backward searching are almost all we need. Consider the standard binary search algorithm of Fig. 4. Each step requires comparing a prefix of some text suffix $T_{A[i],n}$ with the pattern. We can extract such prefix by following the Ψ values recursively: i, $\Psi[i]$, $\Psi[\Psi[i]]$, ..., as they point to $T_{A[i]}$, $T_{A[i]+1}$, $T_{A[i]+2}$, ... 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 $j = i$, $\Psi[i]$, $\Psi[\Psi[i]]$, ..., 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 $C[c] < j \leq C[c+1]$. Thus each $T_{A[j]}$ can be found by an $O(\log \sigma)$-time binary search on C.

The overall time complexity for counting queries is therefore the same $O(m \log n)$ as in the standard binary search multiplied by the time needed for computing one Ψ value and discovering the corresponding text character. As we have described it, this yields $O(m \log n \log \sigma)$ time, yet we show in Section 8 how a simple *rank*-based trick reduces it to the same $O(m \log n)$ time used by the standard binary search. For locating and displaying the text, one can use the same sampling method as in the backward search-based self-indexes.

4.5 Inverted Indexes as Compressed Self-Indexes

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

Consider text `"to be or not to be"`. An inverted index for the 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. It is then easy to implement the search for the occurrences of a query word (e.g. by binary on the vocabulary, or using a trie on the vocabulary words). However, only the word beginnings are indexed, not all substrings as in full-text indexes. We can think of the alphabet of the text as $\Sigma = \{$`"be"`,`"not"`,`"or"`,`"to"`$\}$. Then we have σ increasing occurrence lists to compress (exactly as we will have in function Ψ later on).

An efficient way to compress the occurrence lists is to use variable-length encoding for the gaps, such as Elias-δ coding [Elias 1975; Witten et al. 1999]. Take for example the list for `"be"`: $4, 17$. First, we get smaller numbers by representing the list using differences between adjacent elements (called gaps): $4, (17 - 4) = 4, 13$. The

binary representations of the numbers 4 and 13 are 100 and 1101, respectively. However, from sequence 1001101 one cannot 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 ℓ (= 2) in unary. The next ℓ-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 until the whole bit stream is processed. 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 $\sum_{i=1}^{n_w} I_w[i] \leq n$. The space requirement of the inverted index is then

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

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 equally distributed). We have introduced H_0 as a shorthand notation for the 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 prodced 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 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, and moreover the text could be reconstructed from the index. Although technically this makes this inverted index a self-index, 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 each $\Psi[i]$ value can be computed in $O(\log n)$ time. The absolute samples take $O(n)$ additional bits.

With the help of some small auxiliary tables, the extraction of Ψ values can be carried out in constant time. Self-indexes based on the Ψ function will be studied in more detail in Section 8.

4.6 Roadmap

At this point the reader can leave with a reasonably complete and accurate intuition of the main 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 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 believe the space complexity claims in the rest of the paper. They will have to return to this section only ocassionally 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 one such self-index.

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 Ziv-Lempel 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, tough 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 k-th order empirical entropy of T is reflected in regularities that appear on its suffix array A. In this section we show how some subtle measures of suffix array regularities are related to measures of text compressibility. Those relationships are used later to compress suffix arrays.

The analytical results in this section are presented at a basic level only. We refer the reader to the original sources to find the detailed proofs. Moreover, we

sometimes deviate slightly from the original definitions, changing 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 (actually, the same formula of Manzini [2001] is used by Grossi et al. [2003], yet it is interpreted in this latter sense). 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 algorithm 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*

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

where n_c is the number of occurrences of character c in T. The sum includes only those characters c that occur in T, so that $n_c > 0$.

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

$$H_k \quad = \quad H_k(T) \quad = \quad \sum_{s \in \Sigma^k} \frac{|T^s|}{n} \, H_0(T^s), \tag{1}$$

where T^s is the subsequence of T formed by all the characters that occur followed by the context s. 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 $T^s = T_{p_1-1}T_{p_2-1}\ldots$. Again, we consider only contexts s that do occur in T.

We note that, 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, one can always work on the reverse text if this is an issue. Moreover, it has been shown that the difference is relatively small [Ferragina and Manzini 2005].

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 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 $k \geq 1$. The technical consequence of the finiteness of the text is the presence of the "$" terminator.

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

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. 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"` belong to 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 Z such that $X < Z < Y$ 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 $[j, j+\ell]$ such that $A[j+r] = A[i+r]+1$ for all $0 \leq r \leq \ell$. For technical convenience, cell $A[1] = n$ is considered as a self-repetition of itself, of length 1.*

Note that $A[1] = n$ because $T_{n,n} = \$$, which is lexicographically the smallest suffix and hence it is pointed from $A[1]$.

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 *such that* $A[j_s + r] = A[i_s + r] + 1$ *for all* $0 \le r \le \ell_s$ *(except for* $i_1 = 1$*, where* $A[i_1] = n$*, and thus* $\ell_1 = 0$ *and* $j_1 = 1$*).*

Self-repetitions are best highlighted through the definition of function Ψ, 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 $\Psi : [1, n] \to [1, n]$ *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 $A[\Psi(i)]$ must occur before $A[\Psi(i + 1)]$ in A, that is, $\Psi(i) < \Psi(i + 1)$.

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. Fig. 8 illustrates (for now consider only the first two arrays). This motivates the definition of *runs in* Ψ [Mäkinen and Navarro 2004a, 2005a, 2005b].

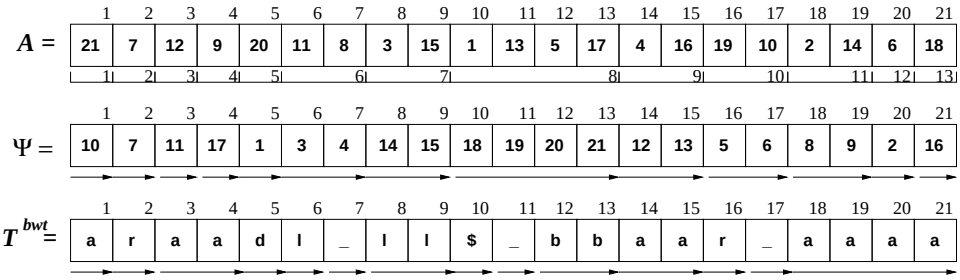

Fig. 8. Covering the suffix array A of the text T = `"alabar a la alabarda$"` with self-repetitions, and the corresponding Ψ function. Below array A we show the minimal covering with self-repetitions, and below array Ψ we show the runs. Both coincide. On the bottom, the characters of T^{bwt}, where we show the equal-letter runs. Almost all *targets* of self-repetitions become equal-letter runs.

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.

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. 8 illustrates. Although the transformation does not compress the text, the transformed text is easier to compress by local optimization methods [Manzini 2001].

We show now an alternative form to regard the BWT, which is useful to understand how it is reversed and to understand several self-indexes presented later.

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 character "$", 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, and L is the list of characters preceding each suffix, that is, $L = T^{bwt}$. On the other hand, F is a sorted list of all the characters in T.

Fig. 9 illustrates. Note that row $M[i]$ is essentially $T_{A[i],n}$ of Fig. 5, and that every column of M is a permution of the text. For compression purposes, it would seem that one could choose the most regular of those permutations to obtain best compression. That would be column F. The problem is that the original text cannot be revealed by just knowing F. However, permutation L can be reversed back to the text, and it also exhibits some compressibility information that can be exploited in many ways, as we show next.

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].

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. 9. 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

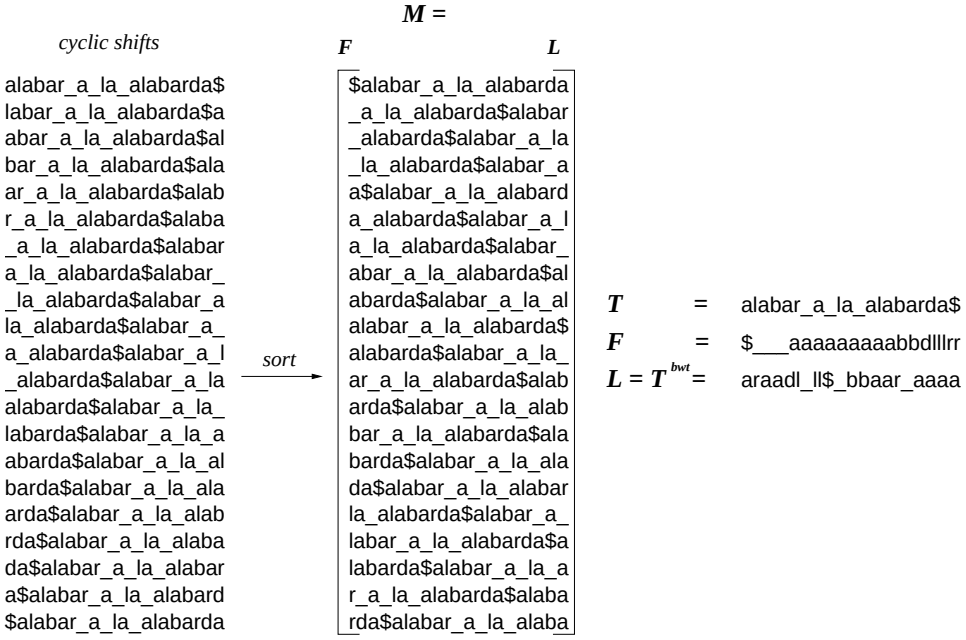

Fig. 9. Obtaining the BWT for the text `"alabar a la alabarda$"`.

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 $L[i] = T^{bwt}_i = T_{A[i]-1}$, or which is the same, the position in A where the suffix $T_{A[i]-1,n}$ is pointed to. 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.

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, the k-th order empirical entropy of T. Note that, for each text context of length k, all the suffixes starting with that context appear consecutively in A. Therefore, the characters that precede each context appear consecutively in T^{bwt}. The following lemma [Ferragina and Manzini 2004; Ferragina et al. 2004a] 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, with k-th order empirical entropy H_k, 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[1], then the representation for the whole T^{bwt} requires at most $cnH_k + \sigma^k f(n/\sigma^k)$ 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, it is enough to encode each portion of T^{bwt} with a zero-order compressor in order 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 a Burrows-Wheeler transformed text T^{bwt}, 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 relationship between the number of runs in Ψ (or self-repetitions in A), and the number of 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 $n_{sr} \leq n_{bw} \leq n_{sr} + \sigma$.*

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

To see why the lemma holds, consider Figs. 3 (page 9) and 8. 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. 8, 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.

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, with k-th order empirical entropy H_k, and given its Burrows-Wheeler transformed text T^{bwt} with n_{bw} equal-letter runs, it holds $n_{bw} \le nH_k + \sigma^k$ for any $k \ge 0$. In particular, it holds $n_{bw} \le nH_k + o(n)$ for any $k \le \log_\sigma n - \omega(1)$. The bounds are obviously valid for $n_{sr} \le n_{bw}$ 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 is the sum of zero-order entropies over all the contexts. It can then be shown that, within a single context $T^{bwt}_{i,j}$, the number of equal-letter runs in $T^{bwt}_{i,j}$ can be upper bounded in terms of the zero-order entropy of the string $T^{bwt}_{i,j}$. 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) $B_{1,n}$, $B_i \in \{0,1\}$ for $1 \le i \le n$, and we want to compress it while at the same time supporting several operations on it. Typical operations are the following:

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 $rank_1(B,i)$ will be 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. Representation is *partial* if it supports the listed operations only for 1-bits, that is, 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 a 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 the $n + o(n)$ solutions 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 $\ell = o(n/\log n)$ 1-bits. One can store all answers explicitly using $O(\ell \log n) = o(n)$ 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 $\lceil \log \binom{n}{\ell} \rceil + o(\ell) + O(\log\log n)$ bits, where ℓ is the number of 1-bits in B. In principle this representation supported only B_i queries, yet it also supported *rank* queries for sufficiently dense bit vectors, $n = O(\ell \operatorname{polylog}(\ell))$. Notice that $\log \binom{n}{\ell} = nH_0(B) + O(\log n)$.

This result was later enhanced by Raman, Raman, and Rao [2002], who developed a representation with similar space complexity, $nH_0(B) + o(\ell) + O(\log\log n)$ bits, supporting *rank* and *select*. However, this representation is partial: they support $select_1(B,j)$ but not $select_0(B,j)$, and $rank_1(B,i)$ only when $B_i = 1$. They also provide a new complete representation requiring $nH_0(B) + O(n \log\log n/\log n)$ bits.

The missing piece, therefore, is achieving a complete representation that needs $o(\ell)$ instead of $o(n)$ extra space when ℓ is significantly smaller than n. This, however, seems unlikely, as there is a matching lower bound [Miltersen 2005] that holds for schemes that do not alter B. This leaves open the question on schemes that replace B with a compressed representation, as the one considered in Section 6.2.

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 results extend to non-binary sequences.

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 $rank_0(B,i)$ is obtained automatically 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 $rank_1(\texttt{smallblock}, t+3) = \texttt{smallrank}[\texttt{smallpiece}^1, t] + \texttt{smallrank}[\texttt{smallpiece}^2, 3]$, where $\texttt{smallpiece}^1$ and $\texttt{smallpiece}^2$ are the two halves of `smallblock`.

We could complete the solution by dividing B into *blocks* of length $\log n$ and explicitly storing *rank* answers for block boundaries, in a table $\texttt{boundaryrank}[0, \frac{n}{\log n} - 1]$, such that $\texttt{boundaryrank}[q] = rank_1(B, q \log n)$ for $0 \leq q < \frac{n}{\log n}$. Then, given $i = q \log n + r$, $0 \leq r < \log n$, we have $rank_1(B, i) = \texttt{boundaryrank}[q] + rank_1(B_{q \log n + 1, q \log n + \log n}, r)$. 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 `boundaryrank` is that there are $\frac{n}{\log n}$ blocks in B, each of which requires $\log n$ bits in `boundaryrank`, for n total bits. To obtain $o(n)$ extra space, we build a *superblocks* dictionary $\texttt{superblockrank}[0, \frac{n}{\log^2 n} - 1]$ such that $\texttt{superblockrank}[q'] = rank_1(B, q' \log^2 n)$ for $0 \leq q' < \frac{n}{\log^2 n}$. We replace the blocks dictionary `boundaryrank` with `blockrank`, such that `blockrank` stores *relative* answers, inside the corresponding superblock of `superblockrank`. That is, $\texttt{blockrank}[q] = \texttt{boundaryrank}[q] - \texttt{superblockrank}[\frac{q}{\log n}]$ for $0 \leq q < \frac{n}{\log n}$. It follows that, for $i = q' \log^2 n + r' = q \log n + r$, $0 \leq r' < \log^2 n$, $0 \leq r < \log n$,

$$rank_1(B, i) = \texttt{superblockrank}[q'] + \texttt{blockrank}[q] + rank_1(B_{q \log n + 1, q \log n + \log n}, r),$$

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. 10 illustrates the structure.

Extending the structure to provide constant time *select* queries is more complicated. We explain a simple version here which shares the essential idea of the real solutions [Munro 1996; Clark 1996].

We partition the space $[1, n]$ of possible *arguments* of *select*, that is, the values j of queries $select_1(B, j)$. We cut $[1, n]$ into blocks of $\log^2 n$ arguments. A dictionary

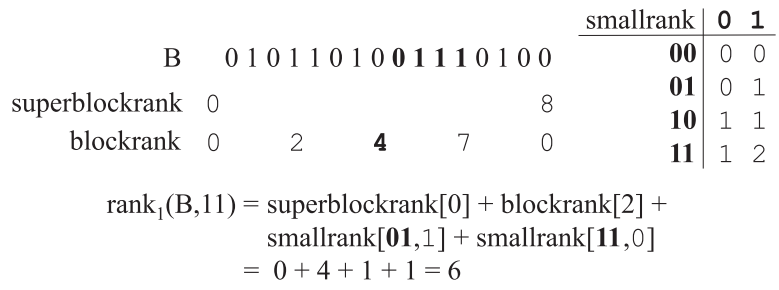

Fig. 10. An example of constant time *rank* computation using $n + o(n)$ bits of space.

`superblockselect`$[j]$, requiring $O(n/\log n)$ bits of space, 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 $n' = \log^2 n$ 1-bits (arguments for $select_1$) and span at most $\log^4 n$ bits in B. We divide them again into *miniblocks* of $\log^2 n' = O((\log\log n)^2)$ arguments. A miniblock directory `miniblockselect`$[j]$ will store the relative answer to $select_1$ inside the short block, that is, `miniblockselect`$[j] = select_1(B, j \log^2 n') -$ `superblockselect`$[\frac{j \log^2 n'}{\log^2 n}]$. 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 $\log^2 n'$ 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. These structures have 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.*

There exist implementations of these solutions which, although not achieving the same theoretical guarantees, work well in practice [González et al. 2005].

6.2 More Sophisticated nH_0-bits 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]. Our presentation is simplified and will only cover $rank_1(B, i)$ queries.

We cut B into blocks of fixed length $t = \frac{\log n}{2}$. Each such block $I = B_{ti+1,ti+t}$

with ℓ bits set will belong to class ℓ 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 ℓ contains $\binom{t}{\ell}$ elements, an index to identify a t-bitmap within its class requires only $\log\binom{t}{\ell}$ bits. Instead of using t bits to represent a block, we use two components (ℓ, r): a class identifier $0 \leq \ell \leq t$, using $\lceil \log(t+1) \rceil$ bits, and an index r within that class, using $\lceil \log\binom{t}{\ell} \rceil$ bits. Then B is represented as a sequence of $\lceil n/t \rceil$ such descriptions.

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 I^i be the i-th block, with ℓ_i bits set. Let us also call ℓ_B the total number of bits set in B. The number of bits required by all the blocks $I^1 \ldots I^{n/t}$ is [Pagh 1999]

$$\left\lceil \log\binom{t}{\ell_1} \right\rceil + \ldots + \left\lceil \log\binom{t}{\ell_{\lceil n/t \rceil}} \right\rceil \quad \leq \quad \log\left(\binom{t}{\ell_1} \times \ldots \times \binom{t}{\ell_{\lceil n/t \rceil}} \right) + n/t$$

$$\leq \quad \log\binom{n}{\ell_1 + \ldots + \ell_{\lceil n/t \rceil}} + n/t \quad = \quad \log\binom{n}{\ell_B} + n/t \quad \leq \quad nH_0(B) + O(n/\log n),$$

where the second inequality holds because the ways to choose ℓ_i bits from each block of t bits are included in the ways to choose ℓ_B 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 (ℓ, 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 obtain the bitmap I from its description. For each class ℓ we store an array $\texttt{decode}_\ell$, so that $\texttt{decode}_\ell[r]$ is the bitmap with index r of class ℓ (in our example, $\texttt{decode}_2[4] = 1001$). Thus, instead of $\texttt{smallrank}[I, i]$ as in Section 6.1, we use $\texttt{smallrank}[\texttt{decode}_\ell[r], i]$, as I is not directly available but just its description (ℓ, r). Table `decode` needs overall $t2^t = O(\sqrt{n} \log n)$ bits. (The real solution does not use `smallrank` but stores the answers to queries directly in `decode`, instead of the bitmaps.) Thus all the extra structures still require $o(n)$ bits. Fig. 11 illustrates.

For *select*, the solution is again more complicated, but it also uses the strategy of dividing blocks into short and long. We omit the details here.

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 access to S, $rank_c$, and $select_c$, in constant time for all symbols c.

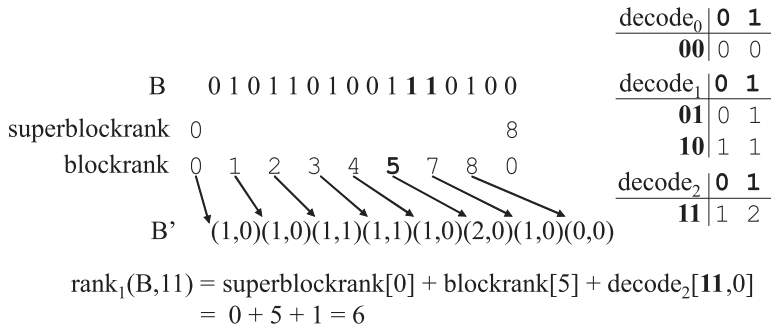

Fig. 11. An example of constant time *rank* computation using $nH_0(B) + o(n)$ bits of space. Here B' is the sequence of pairs (ℓ, r). In a real implementation, this sequence is represented as a binary string, each pair occupying a variable number of bits, and the values in `superblockpos` and `blockpos` point to the corresponding positions in the binary representation of B'. Note also that we have integrated the answers of table `smallrank` into table `decode`, as in the real implementation.

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 $\sum_{c\in\Sigma}(nH_0(B^c)+o(n)) = nH_0(S) + O(n) + o(\sigma n)$ bits.

The $O(n)$ extra term can be removed with a more careful design [Ferragina et al. 2004b, 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. This leads to the next observation.

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.*

This complete representation of S takes sublinear space, namely $o(n \log \sigma)$ bits, only if $\sigma = o(\log n/\log\log n)$.

A completely different technique is the *wavelet tree* [Grossi et al. 2003]. Fig. 6, on page 14, 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 characters in Σ^v. However, sequence S^v is not really stored at the node. Instead, we store a bit sequence B^v telling whether characters in S^v go left or right, that is, $B^v_i = 1$ 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 character $rank_1(B^{root}, i)$ 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. 6 gives an example for $rank_{\texttt{"a"}}(S, 15) = 5$.

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. 6, $select_{\texttt{"a"}}(S, 5)$ starts with the leaf for `"a"`. It is a right child, so in its parent the position is $select_1(1110001101111, 5) = 8$. This in turn is a left child, so in its parent (the root), the final position is $select_0(0100110111001100100000, 8) = 15$.

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. By using r-ary wavelet trees, with suitable r, one can obtain a complete representation with constant-time *rank* and *select* that works for $\sigma = O(\text{polylog}(n))$ [Ferragina et al. 2004b, 2006]. The idea is that, instead of storing bitmaps at each node, we store sequences over an alphabet of size r to represent the tree branch chosen by each character. Those sequences are handled with Lemma 6.

By carefully choosing r, we can get constant access time for $\sigma = O(\text{polylog}(n))$, and improved access time for larger alphabets.

Theorem 9 *Sequence $S_{1,n}$ over an alphabet of size $\sigma = O(\text{polylog}(n))$ can be represented using a multi-ary wavelet tree in $nH_0(S) + O(n / \log^\epsilon n)$ bits of space, for any constant $0 < \epsilon < 1$, so that S_i, $rank_c(S, i)$, and $select_c(S, j)$, can be answered in constant time $O(\frac{1}{1-\epsilon})$.*

Theorem 10 *Sequence $S_{1,n}$ over an alphabet of size $\sigma = o(n / \log\log n)$, can be represented using a multi-ary wavelet tree in $nH_0(S) + O(n\lceil \log \sigma / \log\log n \rceil)$ bits of space, so that S_i, $rank_c(S, i)$, and $select_c(S, j)$, are answered in $O(\lceil \log \sigma / \log\log n \rceil)$ time.*

Finally, we point out some very recent work [Golynski et al. 2006] where they obtain $O(\log\log \sigma)$ time for accessing symbols and *rank*, and constant time for

select, using $n \log \sigma(1 + o(1))$ bits of space.

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 for it, by Chazelle [Chazelle 1988; Kärkkäinen 1999]. For succinctness and simplicity we will focus on the problem variant that appears in the compressed full-text indexes we cover: One has a set of n points over an $n \times n$ grid. 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 by half according to their row value i. Thus it can be implemented 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 bactrack 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 the *abstract optimization* of the suffix array data structure. That is, the search algorithm remains essentially as in Fig. 4, 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.

7.1 Mak-CSA: Mäkinen's Compact Suffix Array

The *Compact Suffix Array* of Mäkinen [2000, 2003] (Mak-CSA) was aimed at explicitly exploiting the self-repetitions of the suffix array A in order to store it in

a more compact form. It was conceived as a succinct index. Only later [Mäkinen and Navarro 2004a] it was shown that its size was related to the text entropy, thus becoming a compressed index. We describe here a version similar in spirit to the original proposal and equally efficient, yet somewhat cleaner.

The idea of the MAK-CSA is, essentially, to represent A as the minimal sequence of the n_{sr} self-repetitions covering A (recall Definition 9). Assume that the partition is $A[i_1, i_1 + \ell_1]$, $A[i_2, i_2 + \ell_2]$, and so on, so that $i_{s+1} = i_s + \ell_s + 1$, $i_1 = 1$, $i_{n_{sr}} + \ell_{n_{sr}} = n$. Let j_s be so that $A[j_s, j_s + \ell_s] = A[i_s, i_s + \ell_s] + 1$. Let p_s be the partition number where j_s lies, that is, $i_{p_s} \leq j_s < i_{p_s+1}$, and let $o_s = j_s - i_{p_s}$ be the corresponding offset. Then, the MAK-CSA for A is an array of n_{sr} *blocks* of the form $B_s = (p_s, o_s, \ell_s, A[i_s])$. That is, for each interval $[i_s, i_s + \ell_s]$, B_s tells the partition where the interval repeats, the offset from the origin of that partition, the length of the repetition, and the explicit A value of the first cell of the block. Sequence B_s contains enough information to reconstruct A.

Fig. 12 illustrates the MAK-CSA for our example text. For example, $B_9 = (8, 2, 1, 4)$ means that block 9 (that is, $A[14, 15] = \{4, 16\}$) is obtained by copying 2 $(1 + 1)$ cells starting at offset 2 of block 8 (and subtracting 1 from them), and it also records $A[i_9] = A[14] = 4$. Note that a pointer from a block may span several other blocks (such as B_8), and that it may point at the middle of a block (as does B_9).

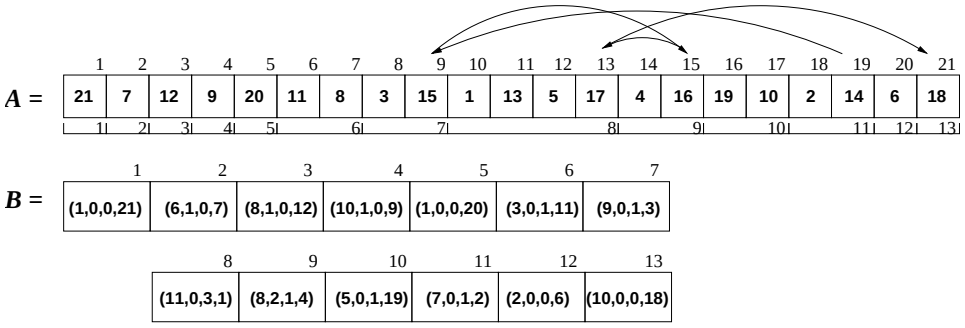

Fig. 12. The MAK-CSA of the text `"alabar a la alabarda$"`. We show the suffix array A and the MAK-CSA B. The minimal covering of self-repetitions is shown on the bottom of A, each region being an entry in B. We also show graphically some pointers related to self-repetitions.

The figure also illustrates partially how we can *decompress* the second cell of block B_{11}. The entry leads us to decompress the second cell of B_7, then the second cell of B_9, then the fourth cell of B_8, and finally the first cell of B_{13}. First cells are explicitly known, in this case its value is 18. Since we made 4 steps to reach B_{13}, the value we wanted is $18 - 4 = 14$. This path must finish at some moment because B_1 is at the end of any path and it is of length 1.

Fig. 13 gives the pseudocode to decompress a cell value. The value $i = i_s + \delta$ we wish to obtain is represented by *reference pair* (s, δ). Note that in lines 1–3 we consider the possibility that $\delta > \ell_s$, in which case the pair (s, δ) is said to be not *normalized*, so we advance in B until finding the correct normalized reference pair. This *normalization* process is necessary because, even if we ensure $\delta \leq \ell_s$ in the

first call, recursive calls with offset $o_s + \delta$ may need normalization. Note that the search finishes when the offset $\delta = 0$, as we explicitly store all first cells of blocks.

```
Algorithm Mak-CSA-decompress(s, δ, B)
(1) while δ > ℓ_s do
(2)     δ ← δ − ℓ_s − 1
(3)     s ← s + 1
(4) if δ = 0
(5)     then return A[i_s];
(5)     else return Mak-CSA-decompress(p_s, o_s + δ, B);
```

Fig. 13. Algorithm to obtain $A[i_s + \delta]$ from the MAK-CSA structure.

Although one could implement a basic binary search (Fig. 4) using B, it is more clever to first binary search on the first cells of the blocks, which are explicitly stored. After $O(\log n_{sr})$ steps, one knows that the answer lies within B_s for some s. At this point, one continues the binary search inside B_s, this time using algorithm Mak-CSA-decompress() to obtain each needed value in $A[i_s, i_s + \ell_s]$. Thus the total cost of the binary search is $O(m \log n)$ as in the basic suffix array, plus that of decompressing some cells of a B block.

The time to decompress a cell is limited by using two parameters C and D. The former is the maximum length of a block, $\ell_s \leq C$, and it is enforced by cutting longer self-repetitions into several ones. Parameter D limits the length of a chain of successive self-repetitions, so as to ensure that we will reach an explicit cell after D recursive invocations of Mak-CSA-decompress(). This is ensured by forcing explicit cells at the D-th position in any self-repetition path. With these parameters, the extra cost of the binary search inside a block is $O(CD \log C)$ (the C factor comes from normalizing the references).

For locating queries, we must decompress several cells of B to obtain the whole interval $A[sp, ep]$. In the worst case this may require a full decompression for each of the *occ* occurrences, at $O(CD\ occ)$ cost.

Given Theorem 5, it is clear that the size of the MAK-CSA is $O(n_{sr} \log n) = O(nH_k \log n)$ bits, thus it is a compressed index. Yet, it is not a self-index, as it still needs T to operate (hence the extra $n \log \sigma$ term in the space complexity that follows). Forcing a maximum block length C introduces at most n/C extra blocks. The same occurs by cutting referencing paths at depth D: every new cut introduced correctly bounds D paths, so n/D cuts suffice. The following theorem assumes $C = D = \log n / \log \log n$ so that the extra space associated to C and D is $O(n \log \log n)$, but other tradeoffs are possible. In addition, this makes each entry of B require only $\log n + 2 \log \log n$ bits (as o_s and ℓ_s are $O(\log n)$).

Theorem 11 (Mäkinen [2003]) *The* Compact Suffix Array *(*MAK-CSA*) offers the following space/time tradeoff.*

Space in bits	$2nH_k \log n + O(n \log \log n) + n \log \sigma$
Time to count	$O(m \log n + \log^2 n / \log \log n)$
Time to locate	$O(\log^2 n / (\log \log n)^2)$
Time to display ℓ chars	$O(\ell)$ (text is available)

In practice. The implementation of MAK-CSA follows closely the description above, except that the blocks are decompressed in pieces taking advantage of the common part of the search path of consecutive elements. This does not improve the worst case complexity, but constitutes a significant speedup in practice.

7.2 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 efficient access to $A[i]$ without representing A, so that the search algorithm is exactly as in Fig. 4. The text T is maintained explicitly.

The GV-CSA uses a hierarchical decomposition of A based on the Ψ function (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. 14 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.)

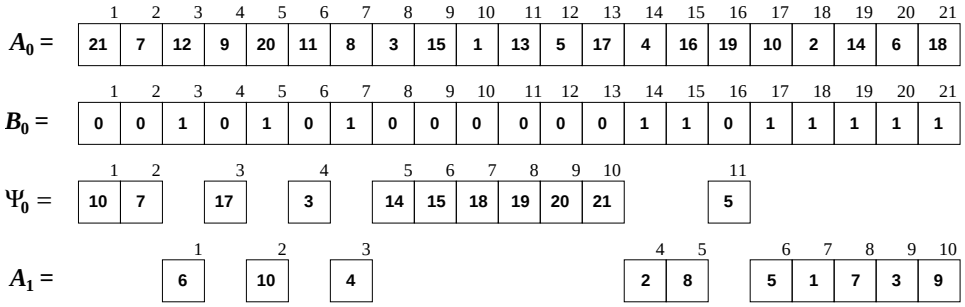

Fig. 14. The first level of the GV-CSA recursive structure, for the text `"alabar a la alabarda$"`. We show the original suffix array $A = A_0$ and structures B_0, Ψ_0, and A_1. We show Ψ_0 and A_1 in scattered form to ease understanding but they are obviously stored contiguously. Recall that A_0 is 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. Fig. 15 gives the pseudocode to extract an entry $A[i]$ from the recursive structure. The complexity is clearly $O(h)$ assuming a constant-time implementation for *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;

Fig. 15. Algorithm to obtain $A[i]$ 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 $H_0^{(2^\ell)} \leq 2^\ell H_0$ [Sadakane 2003], Ψ_0 requires $|T'|H_0^{(2)} + O(|T'| \log \log(\sigma^2)) \leq (n/2)(2H_0) + O((n/2)(1+\log \log \sigma))$. 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 values are inserted every $\Theta(\log n)$ bits, so this adds $O(n)$ bits. To extract an arbitrary position of Ψ, we go to the nearest absolute sample before that position and then sequentially advance summing up differences until reaching the desired position. 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 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 $0 < \epsilon < 1$, can be obtained as follows. Given the $h = \lceil \log \log n \rceil$ levels, we only keep three of them: level 0, level $\lfloor h/2 \rfloor$, and level 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 using the search algorithm of Fig. 4 and the usual way to locate occurrences with A, we get the following results.

Theorem 12 ([Grossi and Vitter 2000; Sadakane 2000]) *The* Compressed Suffix Array *(*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 for all	$0 < \epsilon \leq 1$ *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 and $(1 + 1/\epsilon) n \log \sigma$ bits for the second version of the theorem. 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 $O(h)$).

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 are more interested in 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 $O(nt \log^{1/t} n)$ bits and retrieve $A[i]$ in $O(t)$ time, for any $1 \leq t \leq \log\log n$.

8. TURNING COMPRESSED SUFFIX ARRAYS INTO SELF-INDEXES

Further development over the compressed suffix arrays of Section 7 lead to self-indexes, which can operate without the text. The idea is to replace T by an abstract data type that gives access to any substring of it. 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).

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 $T_{A[i],n}$. With this value one can still use the basic search algorithm of Fig. 4. The SAD-CSA represents A and T using the full function Ψ (Definition 11), and a special representation of function C (Lemma 3). We have already overviewed the basics of this technique in 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 $A[i]$ (or, alternatively, $T_{A[i]} = F[i]$ in Fig. 9), we have that $T_{A[i]} = c$ such that $C(c) < i \leq C(c+1)$ (assuming $C(\sigma+1) = n$). 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 find quickly the c such that $C(c) < i \leq C(c+1)$, we represent C with a bit vector $D[1,n]$, so that $D[i+1] = 1$ iff $C(c) = i < n$ for some $c \in [1,\sigma]$, and a string S where the (at most σ) distinct characters of T are concatenated in alphabetical order (once can also regard D as marking the points where sequence $F[i]$ changes). Therefore, the c such that $C(c) < i \leq C(c+1)$ is precisely $c = S[rank_1(D,i)]$. Using the succinct data structures of Section 6.1 we can compute c in constant time using only $n + o(n)$ bits for D and $\sigma \log \sigma$ bits for S.

Fig. 16 illustrates. To obtain, say, $T_{A[11],n}$ we see that $S[rank_1(D,11)] = S[3] =$ `"a"`. Then we move to $19 = \Psi(11)$. The second character is thus $S[rank_1(D,19)] = S[6] =$ `"l"`. Then we move to $9 = \Psi(19)$ and get third character $S[rank_1(D,9)] = S[3] =$ `"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, that is, 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. Fig. 17 gives the pseudocode to compare P against a suffix of T.

Right now we have used $n + o(n) + \sigma \log \sigma$ bits of space for D and S, plus the representation for Ψ described in Section 7.2, $nH_0 + O(n \log\log \sigma)$ bits. Note that, since the SAD-CSA does not give direct access to $A[i]$, it needs more structures to

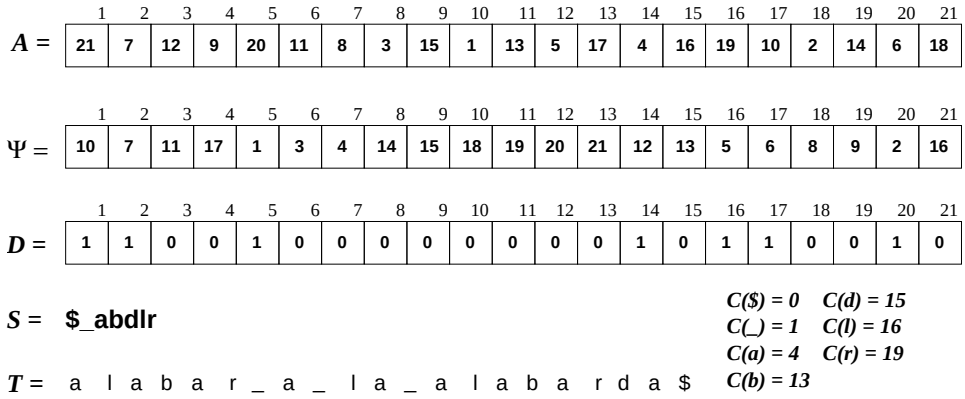

Fig. 16. The main components of the SAD-CSA structure, for the text `"alabar a la alabarda$"`. The text T, the original suffix array A, and function C, are shown for illustrative purposes but are not represented in the structure.

Algorithm Sad-CSA-compare(P, m, i, Ψ, D, S)
(1) $j \leftarrow 1$;
(2) **do** $c \leftarrow S[rank_1(D, i)]$;
(3) **if** $P_j < c$ **then return** "<";
(4) **if** $P_j > c$ **then return** ">";
(5) $i \leftarrow \Psi(i)$;
(6) $j \leftarrow j + 1$;
(7) **while** $j \leq m$;
(8) **return** "=";

Fig. 17. Algorithm to compare P against $T_{A[i],A[i]+m-1}$, where T and A are represented by Ψ, D, and S.

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 the version requiring $\frac{1}{\epsilon} nH_0 + O(n \log\log \sigma)$ bits of space and computing $A[i]$ in $O(\log^\epsilon n)$ time, for any constant $0 < \epsilon < 1$ (Theorem 12).

Sadakane, however, notices an important fact. The same hierarchical GV-CSA structure can serve to compute the inverse of A, $A^{-1}[j]$, within the same $O(\log^\epsilon n)$ time, provided we store explicitly A_h^{-1} in the last level (in addition to the explicit A_h stored in the GV-CSA structure). The reason is that, if one knows that $A^{-1}[j-1] = i$, then $A[i] = j-1$, and therefore $A[\Psi(i)] = j$, thus $A^{-1}[j] = \Psi(i) = \Psi(A^{-1}[j-1])$. Iterating, $A^{-1}[j] = \Psi^k(A^{-1}[j-k])$. Hence, to obtain $A_{h/2}^{-1}[j]$, we take the largest $j' \leq j$ that is represented in the last level, $j' = select_1(B_{h/2}, rank_1(B_{h/2}, j))$, and obtain $i' = A_{h/2}^{-1}[j'] = A_h^{-1}[rank_1(B_{h/2}, j)]$ (note that A_h^{-1} is explicitly stored). Now we apply Ψ successively over i' to obtain $A_{h/2}^{-1}[j] = \Psi_{h/2}^{j-j'}(i')$. To compute $A^{-1}[j] = A_0^{-1}[j]$ we do the same using B_0 and Ψ_0, with the exception that $A_{h/2}^{-1}$ is

not explicitly stored but must be computed with the above procedure.

The inverse of A is useful to implement the additional function that a self-index must provide: given $1 \leq l \leq r \leq n$, retrieve $T_{l,r}$ without having access to T. We already know how to retrieve $T_{A[i],n}$ (or any prefix of it) given i. With the inverse suffix array we first find $i = A^{-1}[l]$ in $O(\log^\epsilon n)$ time, and then retrieve $T_{l,r} = T_{A[i],A[i]+(r-l)}$ in $O(r-l)$ time.

Theorem 13 (Sadakane [2003]) *The* Compressed Suffix Array *(*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*

It is interesting that not only a local portion of T can be decompressed, but also that the suffix array of such a local segment of T can be retrieved using A^{-1} and Ψ.

We note that the method of inserting absolute samples every $\Theta(\log n)$ positions in Ψ and using the four-Russian technique to access between samples in constant time, works with many other compression methods. In particular, if we compress runs in Ψ (Definition 12) with run-length compression (see Section 9.6), we can achieve $nH_k(\log\sigma + \log\log n) + O(n)$ bits of space, for $k \leq \log_\sigma n - \omega(1)$, while retaining the same search times [Mäkinen and Navarro 2004b] (recall Theorem 5).

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 L 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 L give practical tradeoffs between index space and search time.

8.2 MN-CCSA: Mäkinen and Navarro's Compressed Compact Suffix Array

The *Compressed Compact Suffix Array* of Mäkinen and Navarro [2004a] (MN-CCSA) is a simple self-index using essentially the structures of the Mak-CSA of Section 7.1 to implement the search method of the Sad-CSA of Section 8.1. Its main idea is twofold: (1) make the Mak-CSA a self-index, and (2) reduce its space by replacing some of its many pointers by smarter data structures based on *rank* and *select* on bit arrays (Section 6.1).

The MN-CCSA uses the basic structure of the Mak-CSA, with the slight difference that all the suffixes belonging to a single block in the MN-CCSA must start

with the same character, otherwise the block must be split. This produces at most σ extra splits from the optimal covering.

Just as done in Section 8.1, the text is deleted and replaced by bit array D and string S. Yet, since all suffixes in a block start with the same character, we need only one bit in D per block, so $|D| = n_{sr}$. Then, with an equivalent of function Ψ we can retrieve any text string of the form $T_{A[i],n}$ given i. In this case we must keep track all the time of the block number we are at, not only the position in A.

The equivalent of function Ψ is given by array $B_s = (p_s, o_s, \ell_s, A[i_s])$ of Section 7.1. Assume cell i of A corresponds to offset δ in block s, that is, $i = i_s + \delta$ and $0 \leq \delta \leq \ell_s$. Then i is represented by the reference pair (s, δ), and $\Psi(i)$ is represented by reference pair $(p_s, o_s + \delta)$, yet this pair is not necessarily normalized. The MAK-CSA lost $O(C)$ time in normalizing each reference pair before continuing.

The MN-CCSA uses a more compact representation that in addition performs normalization in constant time. Assume we have a bit vector $L[1, n]$ aligned to A, where the block starts are signaled with a bit set. In addition to L, instead of storing sequence $B_s = (p_s, o_s, \ell_s, A[i_s])$ we store the sequence j_s of absolute positions in A where the copy of each block s starts. That is, j_s is the number represented by the reference pair (p_s, o_s). From j_s and L we can obtain other components in constant time: $p_s = rank_1(L, j_s)$, $o_s = j_s - select_1(L, p_s) - 1$, and $\ell_s = select_1(L, p_s + 1) - select_1(L, p_s)$. Furthermore, if current position i is represented by reference pair (s, δ), then $\Psi(i)$ corresponds to normalized reference pair (p_s, δ'), where $p_s = rank_1(L, j_s + \delta)$ and $\delta' = j_s + \delta - select_1(L, p_s)$.

This permits using the same binary search of the SAD-CSA (so we do not give explicit access to $A[i]$ as in the MAK-CSA). Fig. 18 gives the pseudocode.

```
Algorithm MN-CCSA-compare(P, m, i, L, J, D, S)
(1)  j ← 1;
(2)  do  s ← rank_1(L, i);
(3)      δ ← i − select_1(L, s);
(4)      c ← S[rank_1(D, s)];
(5)      if P_j < c then return "<";
(6)      if P_j > c then return ">";
(7)      i ← J[s] + δ;
(6)      j ← j + 1;
(7)  while j ≤ m;
(8)  return "=";
```

Fig. 18. Algorithm to compare P against $T_{A[i],A[i]+m-1}$, where T and A are represented by L, J, D, and S. Array $J[s]$ stores sequence j_s.

Array j_s takes $n_{sr} \log n$ bits, while L takes $n + o(n)$ bits using the techniques of Section 6.1. Likewise, array D takes $n_{sr}(1 + o(1))$ bits. Adding up all the space complexities and recalling Theorem 5, we have overall $n(H_k(1 + \log n) + 1) + o(n)$ bits.

In addition, the MN-CCSA uses $O(n)$ extra bits to locate occurrences and display text contexts. The technique resembles that used for the SAD-CSA in practice (and even more those in the FM-Index [Ferragina and Manzini 2000], Section 9.3). Text positions are sampled at regular intervals $h = \frac{2}{\epsilon} \log n$, and the

positions in A that point to $T_{1,n}$, $T_{h+1,n}$, $T_{2h+1,n}$, and so on, are sorted and stored in array S_p (that is, obtain $A^{-1}[1]$, $A^{-1}[h+1]$, etc., sort them, and store the result in S_p). A bit array $inS_p[1,n]$, aligned to A, tells which positions in A turned out to be sampled in S_p. Then, to find $A[i]$, we take $\Psi(i)$ zero or more times until finding the first r such that $inS_p[\Psi^r(i)] = 1$, at which point we have $A[i] = S_p[rank_1(inS_p, \Psi^r(i))] - r$. This costs $O(\frac{1}{\epsilon}\log n)$ at worst. To show text contexts, the same A positions of S_p are stored in another array S_t, now sorted by their order in T (that is, $S_t[s] = A^{-1}[h \cdot s + 1]$). To display $T_{l,r}$ we compute its immediately preceding sample $s = \lfloor \frac{l-1}{h} \rfloor$, and rather obtain $T_{hs+1,r} = T_{A[S_t[s]],r}$, so we can use the already known method to extract substrings of the form $T_{A[i],n}$. Those structures take $n(1+\epsilon) + o(n)$ extra bits.

Theorem 14 (Mäkinen and Navarro [2004a]) *The* Compressed Compact Suffix Array *(*MN-CCSA*) offers the following space/time tradeoff.*

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

In practice. The MN-CCSA is implemented as described, so it has a major difference with respect to the implementation of the SAD-CSA. The MN-CCSA uses the usual suffix array search method, while the SAD-CSA uses backward searching as explained at the end of Section 8.1. This favors MN-CCSA for large m because the $O(m\log n)$ worst case complexity is no more than $O(\log^2 n)$ on average, while the worst and average case complexities are both $O(m\log n)$ with backward searching, if P appears in T.

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

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

Consider Figs. 3 (page 9) and 16, 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 argument, 4, corresponds to character $T_{A[4]} =$ `" "` preceding `"la"`, while the other two correspond to `"a"` preceding `"la"`.

Fig. 19 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). Seen another way, if $A[i] = j$, then suffix array position i belongs to list T_{j-1} and context $T_{j,j+k-1}$. For example, with $k = 2$, suffix array position 17 ($= \Psi(4)$) points to $T_{A[17]\ldots} = T_{10\ldots}$, so it belongs to list

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. 19. Partition of Ψ into lists (columns) and contexts (rows) in the GGV-CSA, for the text "alabar a la alabarda$". Array Ψ is read from leftmost to rightmost column, each column top to bottom.

$T_9 = T_{A[4]} =$ " " 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 $T_{A[10]+1,A[10]+2} = T_{A[18],A[18]+1} = T_{2,3} =$ "la".

The duality we have pointed out is central to understand what follows. 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 $\Psi(1)$, $\Psi(2)$, ..., $\Psi(i)$, ..., $\Psi(n)$, that is, we are storing vector Ψ. The secondary one is that these Ψ values are indexes to array 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, an order compatible to that of the suffix array. Thus we recover the original sequence Ψ in good order if we read the table column by column (left to right), and each column top to bottom (as we ordered lists and contexts lexicographically). We respect the original suffix array order within cells.

The Ψ values along each list (column) are increasing because Ψ is increasing in the area of A that starts with the same character (Lemma 1). Similarly, Ψ is increasing inside each particular cell.

Consider now for a moment that the numbers correspond to indexes j of A, that is $j = \Psi(i)$ is now seen as an index of A rather than as a value in array Ψ. The row where each j value lies corresponds to the context its pointed suffix starts with, $T_{A[j],A[j]+k-1} = T_{A[\Psi(i)],A[\Psi(i)]+k-1}$.

Take a specific row corresponding to context s. All the j found there are those such that $T_{A[j],A[j]+k-1} = s$. Thus, the j values found in a row form a contiguous subinterval of $[1, n]$ (that is, of A). Each cell in the row corresponds to a different character preceding the context (that is, to a different column), and values are increasing inside each cell.

If we identify each Ψ value in the row with the character of the column (list) it belongs to, then the set of all characters form precisely T^s (Definition 8). Thus, if we manage to encode each row in $|T^s|H_0(T^s)$ bits, we will have nH_k bits overall (recall Eq. (1) and Theorem 4). In our previous example, considering context $s =$ "la",

we have to represent all the Ψ values that lie inside that context ($[17, 19]$) in space proportional to the zero-order entropy of the first characters of the positions in Ψ that point there (in this case, $T_{A[4]}$, $T_{A[10]}$, and $T_{A[11]}$), overall $T^s = T^{bwt}_{17,19} =$ `" 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 $D[1, n]$ of Fig. 16 (corresponding to G_k of Grossi et al. [2003]) suffices, as $c = rank_1(D, i)$ (for clarity we are using c as a character but in practice it is a column number). Using the techniques of Section 6.2, D can be represented in $nH_0(D) + o(n) \leq \sigma \log n + o(n)$ 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 $i' = i - select_1(D, c) + 1$. In the example, to retrieve $\Psi(10) = 18$ (thus $i = 10$), we find $c = rank_1(D, 10) = 3$ (third column in the table, symbol `"a"`). Inside the table, we want the 6th value, because $i' = 10 - select_1(D, 3) + 1 = 10 - 5 + 1 = 6$.

A similar technique gives the right cell within the column. Two bit arrays L_c and E_c are maintained for each column c (these correspond to L^y_k and b^y_k of Grossi et al. [2003]). L_c is aligned to the area of A where suffixes start with character c (appearing in T). L_c contains a 1 every time we change context as we read the numbers in column c, or which is the same, where some of the first $k+1$ characters of the suffixes pointed from the corresponding area in A change. In our example, $L_{\mathtt{a}} = 111101011$, which is aligned to $A[5, 13]$. $L_{\mathtt{a}}[4] = 1$ because $T_{A[8],A[8]+2} =$ `"aba"` $\neq T_{A[7],A[7]+2} =$ `"a l"`; whereas $L_{\mathtt{a}}[5] = 0$ because $T_{A[9],A[9]+2} =$ `"aba"` $= T_{A[8],A[8]+2}$.

E_c, instead, stores one bit per context indicating whether the cell for that context is nonempty in column c. In our example, $E_{\mathtt{a}} = 1110000010111$.

It is easy to see that the relative index i' within column c corresponds to the r'-th nonempty cell of column c, where $r' = rank_1(L_c, i')$. Moreover, the r' nonempty cell has global row number $r = select_1(E_c, r')$. Thus both arrays permit obtaining the row number r for relative index i'. Finally, the position we want inside the r'-th nonempty cell is $p = i' - select_1(L_c, r') + 1$. In our example, the cell is at the 5th nonempty row, as $r' = rank_1(L_{\mathtt{a}}, 6) = 5$. Its global row number is $r = select_1(E_{\mathtt{a}}, 5) = 11$. Its position within the cell is the first, as $p = 6 - select_1(L_{\mathtt{a}}, 5) + 1 = 1$.

Using again the techniques from Section 6.2, each L_c can be stored in $n_c H_0(L_c) + o(n_c)$ bits (where 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 L_c, the space is at most $\sigma^k \log(n_c) + o(n_c)$ bits. Added over all columns, this is at most $\sigma^{k+1} \log n + o(n)$. In the real structure [Grossi et al. 2003], all vectors L_c are concatenated for technical reasons we omit. All the E_c vectors, in turn, using Section 6.1, require $\sigma^{k+1}(1+o(1))$ bits of space.

Finally, we are inside a cell and know which position we seek inside it. Let us regard again the numbers as indexes $j = \Psi(i)$ in A. We need to know which is the range of suffix array positions 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 R (corresponding to F_k of 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 L_c bit vectors (which records context changes within

column c). In our example, $R = 11011110010011010110011$. If we know we are in global row r, then $select_1(R, r)$ tells the first element in the interval handled by row r. In our example, $r = 11$ and $select_1(R, 11) = 17$. We will add this value to the result we obtain inside our cell. Using Section 6.2 once more, R requires $\sigma^k \log n + o(n)$ bits.

The final piece is to obtain the p-th element of cell (r, c) (or locally, the r'-th nonempty cell at column c). At this point there are different choices. One, leading to Theorem 15, is to store a bitmap Z for each cell, indicating which elements of the row interval belong to the cell. This is necessary because any permutation of the A interval of the row can arise among the cells of the row (e.g., see row `"ab"`). In our example, for row 11, the interval is $[17, 19]$, and we have $Z = 100$ at column `" "` and $Z = 011$ at column "a" (zeros and ones can be interleaved in general). With $select_1(Z, p)$ we obtain the offset of our element in the row interval. Therefore, we finally have $\Psi(i) = select_1(R, r) + select_1(Z, p) - 1$. In our example, $select_1(011, 1) = 2$, and therefore $\Psi(10) = 17 + 2 - 1 = 18$. Fig. 20 gives the pseudocode.

Algorithm GGV-CSA-$\Psi(i,\ D,\ L,\ E,\ R,\ Z)$
(1) $c \leftarrow rank_1(D, i)$;
(2) $i' \leftarrow i - select_1(D, c) + 1$;
(3) $r' \leftarrow rank_1(L_c, i')$;
(4) $p \leftarrow i' - select_1(L_c, r') + 1$;
(5) $r \leftarrow select_1(E_c, r)$;
(6) **return** $select_1(R, r) + select_1(Z_{r,c}, p) - 1$;

Fig. 20. Algorithm to compute $\Psi(i)$ using the GGV-CSA.

We consider now the space required by the bit vectors Z. Using again the techniques of Section 6.2, those can be stored in $|Z|H_0(Z) + o(|Z|)$ bits (note that $|Z| = |T^s|$ for each cell in the row corresponding to context s). Summed over all the lists of the same row, this turns out to be $|T^s|H_0(T^s) + |T^s| \log e + o(|T^s|)$ [Grossi et al. 2003]. Added over all the contexts, we get $nH_k + n \log e + o(n)$ (recall Eq. (1) and Theorem 4). To see how the sum of the $H_0(Z)$ entropies over the different columns adds up to $H_0(T^s)$, let us call $z = |Z| = |T^s|$, and z_c the number of entries in column c of row s. Then, recalling the end of Section 5.1, $H_0(Z) \leq z_c \log \frac{z}{z_c} + z_c \log e$, which add up $\sum_{c \in \Sigma} z_c \log \frac{z}{z_c} + z \log e$. But z_c is also the number of occurrences of c in T^s, so the latter is $zH_0(T^s) + z \log e$.

If we add up all the space requirements, we have $nH_k + n \log e + O(\sigma^{k+1} \log n) + o(n)$. For $k + 1 \leq \log_\sigma(n/\log n) - \omega(1)$, the space is $nH_k + n \log e + o(n)$.

Remember that we are not representing Ψ, but Ψ_ℓ for $0 \leq \ell \leq h$ (Section 7.2). 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}. The space requirement at level ℓ turns out to be $nH_k + (n/2^\ell) \log e + O(\sigma^{k+2^\ell} \log n) + o(n)$ bits (remember the space analysis in Section 7.2). To maintain the third term within $o(n)$, it is sufficient that $k + 2^\ell \leq \alpha \log_\sigma n$, for some $0 < \alpha < 1$ of our choice. For this we need two conditions: (1) $k \leq \alpha' \log_\sigma n$ 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 $2^\ell \leq (\alpha - \alpha')\log_\sigma n$ when $\ell \leq h'$. 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 12. The number of steps for that jump is not constant but $2^{h-h'} = O(\log\sigma)$.

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 $nH_k + (n/2^\ell)\log e + o(n)$ each, plus the final level containing A and its inverse. This yields the first result of Grossi et al. [2003].

Theorem 15 (Grossi, Gupta, and Vitter [2003]) *The* Compressed Suffix Array *(*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	$0 < \epsilon \leq 1$ *is an arbitrary constant;*
	$k \leq \alpha\log_\sigma n$, *for some constant* $0 < \alpha < 1$

Note that, compared to Theorem 13, m has been replaced by $m/\log_\sigma n$, and likewise ℓ by $\ell/\log_\sigma n$. This is because they 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 possible to remove. The extra space appears because we are coding each cell individually, indicating in its Z vector which elements of the row interval belong to the cell and which do not. Summed over all the row, the positive information amounts to $|T^s|H_0(T^s) + o(|T^s|)$ bits, but the negative information amounts to the extra $O(|T^s|)$ bits.

An alternative representation for the row is 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 $R_r = T^s =$ `" aa"`. In order to retrieve element p from column c in global row r, we just compute $select_c(R_r, p)$, adding it to $select_1(R, r) - 1$ to return the value in the final line of Fig. 20. In our example, $select_{\texttt{a}}(\texttt{" aa"}, 1) = 2$, which added to $17 - 1$ gives the final answer 18.

The binary wavelet tree requires $|T^s|H_0(T^s) + o(|T^s|)$ bits of space and answers the queries in $O(\log \sigma)$ time. Adding over all the contexts s we get $nH_k + o(n)$ bits. In exchange, the time increases by an $O(\log \sigma)$ factor. In level ℓ, the space remains the same but the query time of the wavelet tree is $O(\log(\sigma^{2^\ell})) = O(2^\ell \log \sigma)$. The search time added over $1 + 1/\epsilon$ levels in $0 \leq \ell \leq h'$ is $O(\log^{1+\epsilon} n)$, while the time to move from level h' to h is $O(\log n \log \sigma)$.

Yet, we still have a problem, namely the extra $O(n)$ bits due to storing A_h and A_h^{-1}. These are converted into $O(n \log\log n / \log_\sigma n)$ by setting $h = \lceil \log\log_\sigma n - \log\log\log n \rceil$. Now the jump from level h' to h takes $O(\log^2 n / \log\log n)$ time. Thus we have an alternative structure.

Theorem 16 (Grossi, Gupta, and Vitter [2003]) *The* Compressed Suffix Array *(*GGV-CSA*) offers the following space/time tradeoffs.*

Space in bits	$\frac{1}{\epsilon} nH_k + o(n \log \sigma)$
Time to count	$O(m \log \sigma + \log^3 n / \log\log n)$
Time to locate	$O(\log^2 n / \log\log n)$
Time to display ℓ chars	$O(\ell / \log_\sigma n + \log^2 n / \log\log n)$
Conditions	$0 < \epsilon \leq 1$ *is an arbitrary constant;*
	$k \leq \alpha \log_\sigma n$, *for some constant* $0 < \alpha < 1$

Note that it is not a good idea to use multiary wavelet trees (e.g. Theorem 9) to improve time, as these limit the alphabet size, while in the last levels of the structure the alphabet reaches size $\sigma^{2^{h'}} = \Theta(n)$.

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 $\frac{1}{\epsilon} nH_k + o(n \log \sigma)$ space, $0 < \epsilon < 1/3$; 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 Z can be represented using run-length encoding and then Elias-γ [Elias 1975; Witten et al. 1999], so that they take in the worst case $2|Z|H_0(Z)$ 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 7.2). 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 $2nH_k$ bits, we could use the same encoding to code each column in Fig. 19 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 shown to be $o(n)$ as long as $k \leq \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$. Thus, we obtain $2nH_k + o(n)$ space. 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. The first exponent of this idea was the FM-Index of Ferragina and Manzini [2000], and many others followed. We start by explaining the idea and then go on to 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 $[sp_m, ep_m] = [C(P_m) + 1, C(P_m + 1)]$ [2]. 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 $[sp_{i+1}, ep_{i+1}]$. Consider again Fig. 9, and the search for $P =$ `"ala"`. Initially, $[sp_3, ep_3] = [C(\texttt{"a"}) + 1, C(\texttt{"b"})] = [5, 13]$. Now, within this subinterval, we wish to know which of those `"a"`s are preceded by `"l"`, as this would give us the occurrences of `"la"`. That is, we know that all the `"l"`s in $L[5, 13]$ appear contiguously in F, and they preserve their relative order. Let s and e be the first and last position in $[5, 13]$ where $L[s] = L[e] =$ `"l"`. In our example, $s = 6$ and $e = 9$. Then, $LF(s)$ and $LF(e)$ are the first and last rows of M that start with `"la"`. In our case, $[sp_2, ep_2] = [LF(6), LF(9)] = [17, 19]$. Recall from Lemma 3 that $LF(6) = C(\texttt{"l"}) + Occ(\texttt{"l"}, 6) = 16 + 1$ and that $LF(9) = C(\texttt{"l"}) + Occ(\texttt{"l"}, 9) = 16 + 3$. With the same mechanism we finally get $[sp_1, ep_1] = [10, 11]$.

In general, given $[sp_{i+1}, ep_{i+1}]$, we determine s and e and then have $[sp_i, ep_i] = [LF(s), LF(e)]$. The problem is that we do not know s and e. Yet, this is not necessary. Since s is the position of the first occurrence of P_i in $L[sp_{i+1}, ep_{i+1}]$, it follows that $Occ(P_i, s) = Occ(P_i, sp_{i+1} - 1) + 1$. Likewise, $Occ(P_i, e) = Occ(P_i, ep_{i+1})$ because e is the last occurrence of P_i in $L[sp_{i+1}, ep_{i+1}]$. The resulting algorithm is rather simple and is shown in Fig. 21.

[2]Here we use $a + 1$ to denote the character that follows a in Σ.

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

Fig. 21. Backward search algorithm to find the interval in $A[1, n]$ of the suffixes that start with $P_{1,m}$.

Note that function C can be implemented in constant time as an array, using just $\sigma \log n$ bits. A similar approach with Occ would require $\sigma n \log n$ bits, which is too much space. All the different variants of the backward search concept aim basically at implementing Occ in little time and 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 the previous section.

9.2 Backward Searching using Ψ

Before entering into the more typical backward search implementations, which are based on the BWT, we show that backward searching can be implemented using function Ψ [Sadakane 2002]. As mentioned at the end of Section 8.1, this is how the SAD-CSA is actually implemented.

Since Ψ and LF are inverse functions, we might binary search values $LF(s)$ 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. A disadvantage is that the average and worst-case costs are similar, because we perform m steps of cost $O(\log n)$ each. In classical searching, we perform $O(\log n)$ steps of cost between 1 and m each, the average case being $O(\log^2 n)$ rather than $O(m \log n)$ if $m > \log n$ (see the end of Section 8.2). On the other hand, Sadakane [2002] 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 9.5.

We also recall (end of Section 8.1) that function Ψ is implemented via sampling, so the binary searching is performed first over the samples and then completed with a sequential decompression plus search between two samples. If the sampling step is $L = \Theta(\log n)$ (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 the normal suffix array searching, there would have been $O(m \log n)$ accesses to 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]. This will be called FMI in this paper.

Ferragina and Manzini [2000] show that Occ can be implemented in constant time, using space upper bounded by $5nH_k + o(n)$. This was the first structure achieving $O(nH_k)$ bits of 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 using results from Manzini [2001] that this representation compresses T^{bwt} to at most $5nH_k + O(\sigma^k \log n)$ bits. The compressed T^{bwt} will be called Z.

The directories to answer $Occ(c, i)$ resemble the solution for *rank* in Section 6.1. We choose a block length ℓ and cut the range $[1, n]$ into *blocks* of length ℓ. Every ℓ consecutive blocks will be grouped into a *superblock* of length ℓ^2. For each superblock $1 \leq i \leq \lceil n/\ell^2 \rceil$ and each character $c \in \Sigma$, we store $NO[c, i] = Occ(c, (i-1)\ell^2)$, the Occ value for character c just before superblock i starts. We also store $W[i]$, the bit position of $T^{bwt}_{(i-1)\ell^2+1}$ (the first character of superblock i) in the compressed representation Z. These two tables require $O((n\sigma \log n)/\ell^2)$ bits. Similarly, for each block $1 \leq j \leq \lceil n/\ell \rceil$ we store the same data relative to its superblock $i = 1 + \lfloor j/\ell \rfloor$, that is, $NO'[c, j] = Occ(c, (j-1)\ell) - Occ(c, (i-1)\ell^2)$, as well as $W'[j]$, the bit position of $T^{bwt}_{(j-1)\ell+1}$ in Z minus $W[i]$. It is easy to see that $NO'[c, j] \leq \ell^2$, and similarly it holds $W'[j] = O(\ell^2 \log \sigma)$, so these two tables require $O((n\sigma \log \ell)/\ell)$ bits. In addition, for each block j, the state of the move-to-front transformation (that is, the recency rank of characters) at the beginning of the block is maintained in $MTF[j]$. This requires $O((n\sigma \log \sigma)/\ell)$ additional bits.

The final component is a table S that does not depend on Z. $S[c, o, B, M]$ is indexed by a character $c \in \Sigma$, an offset $o \in [1, \ell]$ inside a block, the content of a block (a bit stream) whose length is in the worst case $\ell' = (1 + 2 \log \sigma)\ell$, and the state of a move-to-front transformation (a permutation of $[1, \sigma]$). The content of $S[c, o, B, M]$ is $Occ(c, o)$ for the text obtained by decompressing B starting with a move-to-front transform initialized as M.

It is not hard to see that, given position u, we can compute $i = 1 + \lfloor u/\ell^2 \rfloor$, $j = 1 + \lfloor u/\ell \rfloor$ and $o = u - (j-1)\ell$, and then it holds $Occ(c, u) = NO[c, i] + NO'[c, j] + S[c, o, Z_{W[i]+W'[j], W[i]+W'[j+1]-1}, MTF[j]]$ (if $j = \ell - 1$ then $W[i] + W'[j+1]$ should be replaced by $W[i+1]$). This can be computed in constant time on a RAM machine provided $|B| = \ell' = O(\log n)$ and $|M| = O(\log n)$. The first restriction

yields $\ell = O(\log_\sigma n)$, whereas the second becomes $\sigma = O(\log n / \log\log n)$.

Fig. 22 illustrates, with $\ell = 3$. For example, to compute $Occ(\texttt{"a"}, 14)$ we consider superblock $i = 1 + \lfloor 14/\ell^2 \rfloor = 2$, block $j = 1 + \lfloor 14/\ell \rfloor = 5$, and offset $o = 14 - (j-1)\ell = 2$. Then $Occ(\texttt{"a"}, 14) = NO[\texttt{"a"}, 2] + NO'[\texttt{"a"}, 5] + S[\texttt{"a"}, 2, \{1,6,1\}, \texttt{"b \$ldar"}] = 3 + 0 + 1 = 4$.

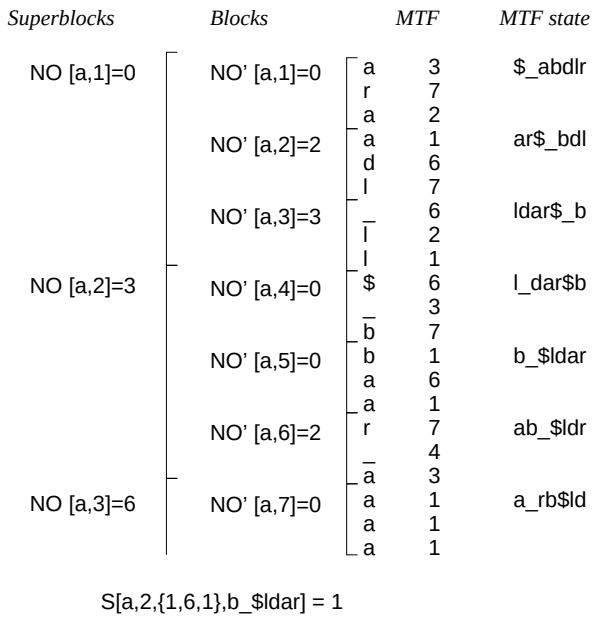

Fig. 22. The main components of the FMI structure, for the text `"alabar a la alabarda$"` using $\ell = 3$. The compression of T^{bwt} is illustrated only up to the move-to-front (MTF) transform for readability, and we also show the MTF states stored per block. We show the NO and NO' values only for character `"a"`, and omit W and W'. We show just a single entry of S.

Adding up all the space complexities we obtain, apart from $|Z|$, the upper bound $O((n\sigma\log n)/\ell^2 + (n\sigma\log\ell)/\ell + (n\sigma\log\sigma)/\ell + \sigma\ell 2^{\ell'}\sigma!\log\ell)$. If we choose $\ell = x\log_\sigma n$ for constant $0 < x < 1/3$, then $\ell' \le 3x\log n < \log n$. Under this setting the extra space is $O((n\sigma\log\sigma\log\log n)/\log n + (\sigma/e)^{\sigma+3/2} n^{3x}\log_\sigma n\log\log n)$. This is $o(n\log\sigma)$ (that is, sublinear in $|T|$) for $\sigma = o(\log n/\log\log n)$.

In order to locate occurrences, they sample text positions at regular intervals, just as described in Section 8.2 (the concept was invented by Ferragina and Manzini [2000] and reused for the MN-CCSA). Instead of marking one text position every $\frac{1}{\epsilon}\log n$, they mark one text position every $\log^{1+\epsilon} n$, for some $\epsilon > 0$, and collect the A values pointing to those marked positions. 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 and can locate the occurrences in $O(occ\log^{1+\epsilon} n)$ steps. A problem is that there is no easy way to know T_i^{bwt} in order to compute $LF(i) = C(T_i^{bwt}) + Occ(T_i^{bwt}, i)$ (Lemma 3). They discover T_i^{bwt} by sequentially searching Σ for the c such that

$Occ(c,i) \neq Occ(c,i-1)$. This takes $O(\sigma)$ time per step. A similar approach permits displaying $T_{l,r}$ in $O(\sigma(r-l+\log^{1+\epsilon} n))$ time.

Theorem 17 (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	$O(\sigma \log^{1+\epsilon} n)$
Time to display ℓ chars	$O(\sigma(\ell + \log^{1+\epsilon} n))$
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 the S table. Ferragina and Manzini [2001] propose replacing S with a plain decompression and scanning of block B, which (according to the theoretical value of ℓ) 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 a table telling which positions are marked, as this 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`, with 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 reason for this lack of guarantee is that T^{bwt} is partitioned into equal-size blocks, not according to contexts of length k. Such a partitioning will be considered in Section 9.7.

9.4 Huff-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. The *Huffman FM-Index* by Grabowski et al. [2004, 2005] (HUFF-FMI) uses Huffman as a tool to reduce the alphabet of the text to bits.

The idea is to first compress T using Huffman [Huffman 1952; Bell et al. 1990]. In the resulting bit stream $T'_{1,n'}$, of $n' < n(H_0+1)$ bits, logically mark the bits that

start a codeword. Apply the BWT over T' to obtain bit stream $B[1,n'] = (T')^{bwt}$. Create another bit stream $Bh[1,n']$ indicating the pointed bits that are marked in T'. That is, if $A'[1,n']$ is the suffix array of T', then $Bh[i] = 1$ iff $T'[A'[i]]$ is marked. No attempt is made to compress B nor Bh.

Implementing Occ over B is very easy, because it is a binary stream. Indeed, $Occ(0,i) = rank_0(B,i)$ and $Occ(1,i) = rank_1(B,i)$, which can be answered in constant time using $n' + o(n')$ bits with the techniques of Section 6.1. Any alphabet dependence has vanished. (We omit some technicalities due to boundary effects because of not having a terminator character for T'.)

To count the occurrences of $P_{1,m}$ in T, we first Huffman-transform P into a binary pattern $P'_{1,m'}$, using the codebook we have created for T. Then, we use the backward search of Fig. 21 for P' over B, finishing after $O(m')$ constant-time steps with $[sp', ep']$ as the bounds in A' for the suffixes that start with P'. Note that $m' < m(H_0 + 1)$ if P has the same zero-order entropy of T, and in any case $m' = O(m \log n)$ because the longest Huffman code has $O(\log n)$ bits [Witten et al. 1999, pp. 397].

Yet, not every occurrence of P' in T' corresponds to an occurrence of P in T, as we want only the codeword-aligned occurrences. This is where Bh comes into play. Only the bits set in $Bh[sp', ep']$ correspond to real occurrences, so we complete the counting query by returning $rank_1(Bh, ep') - rank_1(Bh, sp' - 1)$. In order to find the exact entries in A' corresponding to occurrences, we start with $b = rank_1(Bh, sp' - 1)$ and use $select_1(Bh, b + 1)$, $select_1(Bh, b + 2)$, and so on, taking constant time per occurrence.

Fig. 23 illustrates the search for $P =$ `"ba"`. This is translated into $P' =$ 11010. The backward search yields $A'[43, 46]$. From those, only $rank_1(Bh, 46) - rank_1(Bh, 42) = 2$ are valid.

To locate the occurrences, the same mechanism of the MN-CCSA (Section 8.2) of marking T' at regular intervals of the form $\frac{1}{\epsilon}(H_0 + 1) \log n$ is employed, yet the marks are slightly moved to align to codeword beginnings. The absolute values stored for the samples correspond to positions in T, not in T'. Using the LF-mapping over B and using Bh to keep count of how many positions in T we have moved, we can deduce the text position corresponding to $A'[i]$ in $O(\frac{1}{\epsilon}(H_0+1) \log n)$ time. Finally, to display $T_{l,r}$ we store the same samples, this time in text position order and pointing to A' entries. After extracting the bits from T', we decode them to get $T_{l,r}$.

Theorem 18 (Grabowski, Mäkinen, Navarro, and Salinger [2005]) *The* Huffman FM-Index *(*HUFF-FMI*) offers the following space/time tradeoff.*

Space in bits	$n(2H_0 + 3 + \epsilon) + o(n)$
Time to count	$O(m(H_0 + 1))$ *(average case)*
	$O(m \log n)$ *(worst case)*
Time to locate	$O(\frac{1}{\epsilon}(H_0 + 1) \log n)$
Time to display ℓ chars	$O((H_0 + 1)(\ell + \frac{1}{\epsilon} \log n))$ *(average case)*
	$O((\ell + \frac{1}{\epsilon}(H_0 + 1)) \log n)$ *(worst case)*
Conditions	$\sigma = o(n/\log n)$
	$\epsilon > 0$ *is an arbitrary constant*

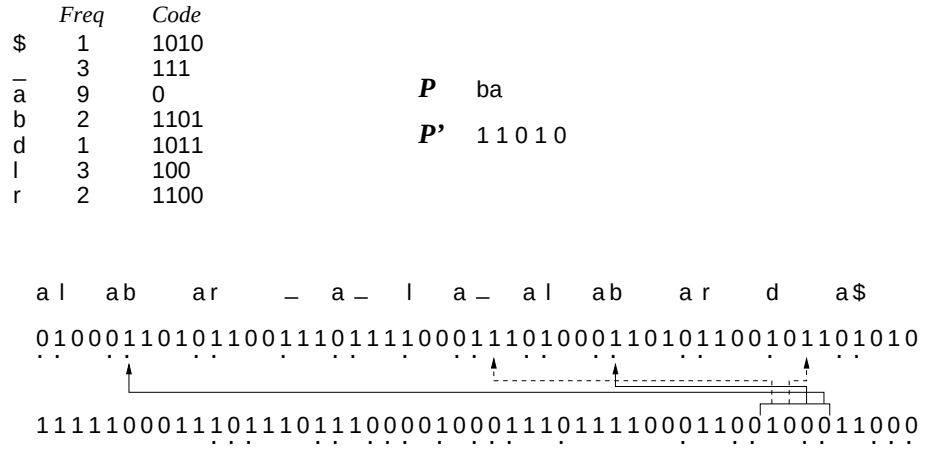

Fig. 23. The main components of the HUFF-FMI structure, for the text `"alabar a la alabarda$"`. We show T', and the dots below its bits indicate those that start a codeword. We also show the bits of B, and those of Bh correspond to the dots below B. The search pattern $P =$ `"ba"` is translated into $P' = 11010$. The backward search yields $A'[43, 46]$, as indicated on top of B. From those, only two are valid (those with dots in the result range). We show explicitly the location of the results, both valid (solid line) and invalid (dashed line).

We note that the average-case complexity for counting assumes that P has the same zero-order entropy of T (in which case it is a worst-case complexity), or that P is chosen randomly from T. The average complexity for displaying text assumes that $[l, r]$ is chosen at random in T. Grabowski et al. [2005] show that, by using $2n$ additional bits, the worst cases can be reduced to $O(m \log \sigma)$ for counting, and $O(\ell \log \sigma + \frac{1}{\epsilon}(H_0 + 1) \log n)$ for displaying. This is obtained by forcing the Huffman tree to have depth at most $(1 + x) \log \sigma$, for some constant $x > 0$, without changing its average codeword depth $(H_0 + 1)$ by more than a constant factor.

In practice: The HUFF-FMI implementation is close to its theoretical proposal. The main difference is that *select* is not used, but the area $Bh[sp', ep']$ is linearly traversed to find the occurrences. This is much faster as on average we have to traverse $H_0 + 1$ bits to find each real answer (this is at most 8 if symbols fit in bytes). In addition, 2^k-ary Huffman has been tried, which improves search time and may increase or decrease the space complexity (as $|B|$ increases slightly due to the less efficient (non-binary) compression; $|Bh|$ is almost divided by k because only positions multiple of k can start a codeword; and one needs 2^k *rank* structures for B).

It is worth mentioning that there is another approach [Ferragina 2006] to combining Huffman compression with the FM-Index, based on using 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 WT-FMI: A Cleaner $O(nH_0)$ Size Implementation

A second idea producing an index of size $O(nH_0)$ is the *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. 2004a] as a particular case of the AF-FMI (Section 9.7), and even later [Mäkinen and Navarro 2005a, 2005b] as a particular case of the RL-FMI (Section 9.6). 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). 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^{bwt}_i in $O(\log \sigma)$ or $O(1)$ time using the same wavelet tree.

Fig. 6 in page 14 illustrates how to obtain $Occ(\texttt{"a"}, 15) = rank_{\texttt{"a"}}(T^{bwt}, 15) = 5$, using a binary wavelet tree. Fig. 24 gives the pseudocode. Depending on which wavelet tree we use, different tradeoffs are obtained. Yet, we have chosen to give a simple general form that is valid for all cases.

Algorithm WT-Occ(c, i, σ_1, σ_2, v)
(1) **if** $\sigma_1 = \sigma_2$ **then return** i;
(2) $\sigma_m = \left\lfloor \frac{\sigma_1+\sigma_2}{2} \right\rfloor$;
(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);

Fig. 24. 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.

Theorem 19 *The* Wavelet Tree FM-Index *(*WT-FMI*) offers the following space/time tradeoffs.*

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	$\sigma = o(n / \log\log n)$*;*
	$\epsilon > 0$ *is an arbitrary constant*

Note that $\log \sigma / \log\log n = O(1)$ *if* $\sigma = O(\text{polylog}(n))$*, in which case, for example, counting time becomes* $O(m)$*.*

Despite its simplicity, the WT-FMI is the precursor of further research that lead to the best implementations of the backward search concept (Sections 9.6 and 9.7).

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 $nH_0 + o(n)$ bits, we obtain $n(H_0 + 1)(1 + o(1))$ 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 it can be limited to $O(m \log \sigma)$ without losing the $O(mH_0)$ average time, by forcing the Huffman tree to balance after depth $(1+x) \log \sigma$, for some constant $x > 0$ [Mäkinen and Navarro 2004b].

9.6 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 of Section 9.5, which exploits the equal-letter runs of the BWT (Theorem 5) to achieve $O(nH_k \log \sigma)$ 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. 6 we built the wavelet tree of $T^{bwt} =$ `"araadl ll$ bbaar aaaa"`. Assume that we run-length compress T^{bwt} to obtain $S =$ `"aradl l$ bar a"`. By Theorem 5, we have the limit $|S| \leq nH_k + \sigma^k$ for any k. Therefore, a wavelet tree (Section 6.3) built over S would require $(nH_k + \sigma^k)H_0(S) + o(n \log \sigma)$ bits. The only useful bound we have for the zero-order entropy of S is $H_0(S) \leq \log \sigma$, thus the space bound is $nH_k \log \sigma + o(n \log \sigma)$ for any $k \leq \log_\sigma n - \omega(1)$.

The problem is that *rank* over S does not give the answers we need over T^{bwt}. For example, assume we want to compute $rank_{\texttt{"a"}}(T^{bwt}, 19) = 7$. We need to know that T^{bwt}_{19} lies at S_{14}. This is easily solved by defining a bitmap $B[1, n]$ indicating the beginnings of the runs in T^{bwt}. In our case $B = 11101111010111010111000$. We know that the position of T^{bwt}_{19} in S is $rank_1(B, 19) = 14$. Yet, this is not sufficient, as $rank_{\texttt{"a"}}(S, 14) = 4$ 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 B alphabetically, accordingly to the characters that form the run. Runs of the same character stay in the same relative order. We form bit array $B'[1, n]$ with the reordered B. In our case $B' = 11111101010000101110 11$. We also compute array C_S indexed by Σ, so that $C_S[c]$ tells the number of occurrences in S (runs in T^{bwt}) of characters smaller than c (thus C_S plays for S the same role C plays for T in the FMI). In our example $C_S[\texttt{"a"}] = 4$. This means that, in B', the first $C_S[\texttt{"a"}] = 4$ runs correspond to characters smaller than `"a"`, and then come those of `"a"`, of which T^{bwt}_{19} is in the 4th because $rank_{\texttt{"a"}}(S, 14) = 4$. Fig. 25 illustrates.

To compute $rank_c(T^{bwt}, i)$, we first find $i' = rank_1(B, i)$, the position of the run T^{bwt}_i belongs to in S. Thus there are $j' = rank_c(S, i')$ runs of c's in $T^{bwt}_{1,i}$. In B', the runs corresponding to c start at $j = select_1(B', C_S[c] + 1)$. Now there are two cases. If $S_{i'} \neq c$, then the run of T^{bwt}_i does not belong to c, and thus we must accumulate the full length of the first j' runs of c, $select_1(B', C_S[c] + 1 + j') - j$. If,

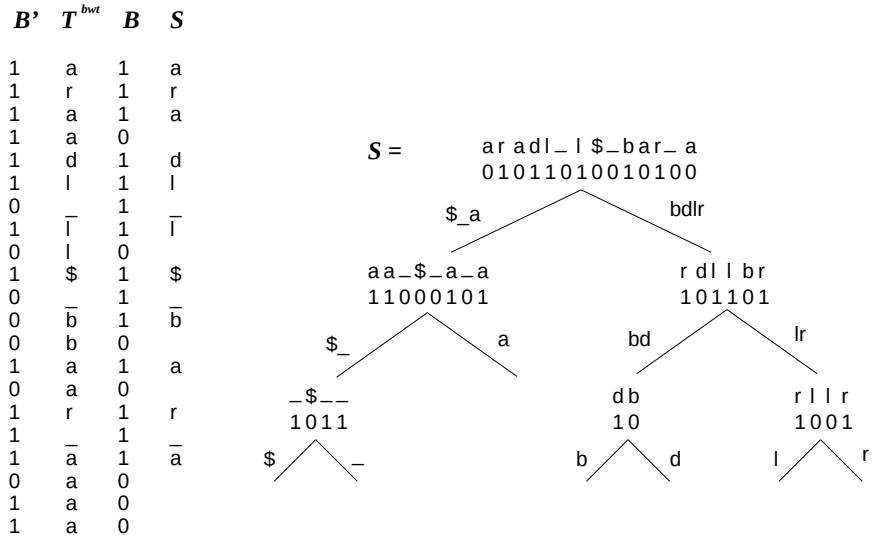

Fig. 25. The main RL-FMI structures for the text `"alabar a la alabarda$"`. The transformed text T^{bwt} is shown only for clarity. The wavelet tree on the right, built for S, stores only the bitmaps, not the texts at each node.

on the other hand, $S_{i'} = c$, then the run of T_i^{bwt} does belong to c, and we must be careful how much of the last run we count. We are sure that the first $j' - 1$ runs must be fully counted, so we have $select_1(B', C_S[c] + j') - j$, but we must add the corresponding part of the last run. This part is clearly $i - select_1(B, i') + 1$. Fig. 26 gives the pseudocode.

Algorithm RLFM-Occ(c, i, S, B, B', C_S)
(1) $i' \leftarrow rank_1(B, i)$;
(2) $j' \leftarrow rank_c(S, i')$;
(3) $j \leftarrow select_1(B', C_S[c] + 1)$;
(4) **if** $S_{i'} = c$ **then**
(5) $\quad j' \leftarrow j' - 1$;
(6) $\quad ofs \leftarrow i - select_1(B, i') + 1$;
(7) **else** $ofs \leftarrow 0$;
(8) **return** $select_1(B', C_S[c] + 1 + j') - j + ofs$;

Fig. 26. Algorithm to compute $Occ(c, i)$ with the RL-FMI.

Thus the RL-FMI takes $nH_k \log \sigma + 2n + o(n \log \sigma)$ bits of space, and it solves $Occ(c, i) = rank_c(T^{bwt}, i)$ in the time necessary to perform $rank_c$ over S. Depending on which wavelet tree is used to implement this operation, we have different results. The rest is handled just like the WT-FMI.

Theorem 20 (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	$\sigma = o(n/\log\log n)$;
	$\epsilon > 0$ *is an arbitrary constant;*
	$k \leq \log_\sigma n - \omega(1)$

In practice. The implementation of the RL-FMI, just as that of the WT-FMI (end of Section 9.5), uses binary wavelet trees with Huffman shape, with the bitmaps using the techniques of Section 6.1. This gives at most $nH_k(H_0(S)+1)(1+o(1))$ space, which in the worst case is $nH_k(\log\sigma+1)(1+o(1))$ bits, close to the theoretical version but much simpler to implement. Even when it has not been proved that $H_0(S)$ is smaller than anything other than $\log\sigma$, in practice space and access time are reduced.

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

The *Alphabet-Friendly FM-Index* of Ferragina, Manzini, Mäkinen, and Navarro [2004a, 2004b, 2006] (AF-FMI) is another improvement over the WT-FMI of Section 9.5. It combines the idea with Theorem 4 to achieve $nH_k + o(n \log \sigma)$ 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 $n_s H_0(T^s) + f(n_s)$ bits, for any convex function f, then the sum of all bits used is $nH_k + \sigma^k f(n/\sigma^k)$. In particular, we can use the binary wavelet tree of Section 9.5 for each block (Theorem 6.3). It requires $n_s H_0(T^s) + O(n_s \log\log n_s / \log_\sigma n_s)$ bits, so we need overall $nH_k + O(n \log\log(n/\sigma^k)/\log_\sigma(n/\sigma^k))$ bits, for any k. If $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$, this space is $nH_k + O(n \log\log n / \log_\sigma n)$.

Assume s is the j-th nonempty context 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 a bit vector $B[1,n]$ indicating the beginning of blocks in T^{bwt}, so that $j = rank_1(B,i)$ (this is equivalent to vector F in Section 8.3). Moreover, the j-th block starts at $i' = select_1(B,j)$.

We also store, for each block j, vector $C_j[c]$, 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 $C_j[c] = Occ(c,i'-1)$, it is clear that $Occ(c,i) = C_j[c] + Occ_j(c, i-i'+1)$. 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.

Using the technique of Section 6.2, bit vector B requires at most $\sigma^k \log n$ bits because it has at most σ^k bits set. On the other hand, arrays C_j require overall $\sigma^{k+1} \log n$ bits of space. If $k \leq \alpha \log_\sigma n$ this space is within $nH_k + o(n)$.

Fig. 27 illustrates, for $k = 1$. To determine $Occ(\texttt{"1"}, 8)$, we first find that $j = rank_1(B,8) = 3$ is the block number where $i = 8$ belongs. The first position of block 3 is $i' = select_1(B,3) = 5$. The number of occurrences of `"1"` in $T^{bwt}_{1,i'-1}$, that

is, before block $j = 3$, is $C_3(\texttt{"l"}) = 0$. Inside block 3, corresponding to substring $T^s = \texttt{"dl ll\$ bb"}$ 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 answers $Occ_3(\texttt{"l"}, 4) = 2$. Thus the answer is $0 + 2 = 2$.

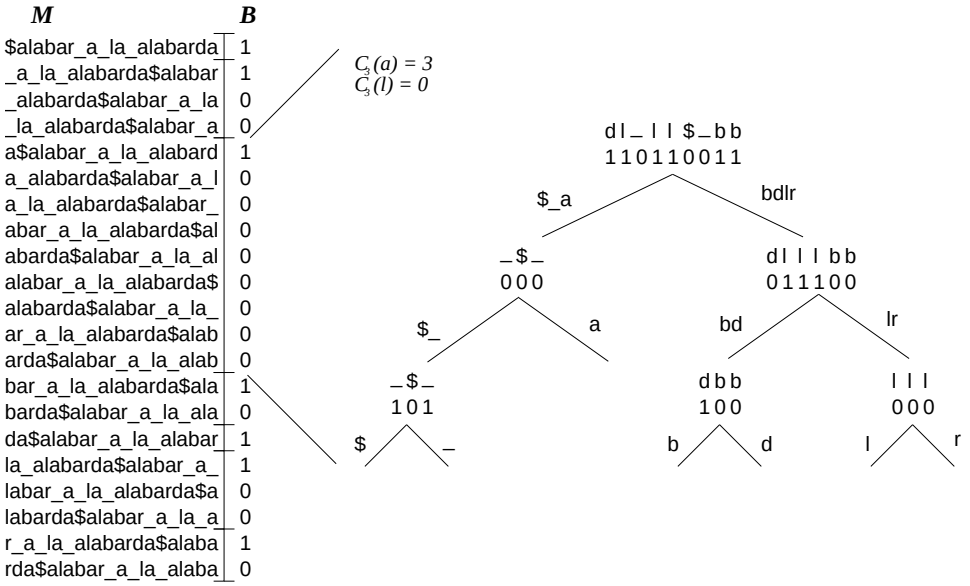

Fig. 27. The main AF-FMI structures for the text `"alabar a la alabarda$"` considering contexts of length $k = 1$. Matrix M of the BWT is shown only for clarity. We show only one of the wavelet trees, corresponding to context `"a"`, and only a couple of its Occ_j values. Note that this wavelet tree corresponds to a substring of that in Fig. 6.

Yet, Ferragina et al. [2004a] go further. Instead of choosing a fixed k value in advance, they use a method by Ferragina and Manzini [2004] that, given a space overhead function $f(n_s)$ on top of $n_s H_0(T^s)$, 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 $nH_k + \sigma^k f(n/\sigma^k)$ *simultaneously for all* k (and it is possibly better than using any fixed k). That is, although we have made the analysis 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 8.3, 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 9.5.

Theorem 21 (Ferragina, Manzini, Mäkinen, and Navarro [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	$\sigma = o(n/\log\log n)$*;*
	$\epsilon > 0$ *is an arbitrary constant;*
	$k \leq \alpha \log_\sigma n$*, for some constant* $0 < \alpha < 1$

10. ZIV-LEMPEL 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 Ziv-Lempel compression. In particular, one of those achieves $O(m + occ)$ time to locate the *occ* occurrences of P in T, with no restriction on m or *occ*. This has not been achieved with other indexes.

10.1 Ziv-Lempel Compression

In the seventies, Ziv and Lempel [1977, 1978] 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. Ziv-Lempel methods produce a *parsing* (or partitioning) of the text into *phrases*.

Definition 16 *The* original Ziv-Lempel 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 $T_{1,i-1}$. If $i' > i$ then $Z[p] = T_{i,i'-1}$ and we continue from text position i'. Otherwise T_i has not appeared before and $Z[p] = T_i$, continuing from text position $i+1$. The process finishes once we obtain $Z[n'] =$ "\$".*

The output of a compressor using this parsing is essentially the position where each new phrase starts (called its *source*) and its length (new characters in Σ that appear are exceptions in this encoding). An important property of this parsing is that every phrase has appeared before, unless it is a new character of Σ. (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 here.)

We will also be interested in a Ziv-Lempel 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 $Z[p']$, $1 \leq p' < p$. Thus $T_{i,n} = Z[p']\, c\, T_{i',n}$, where $Z[p']\, c$ does not appear in Z and $i' = i + |Z[p']| + 1$. We define $Z[p] = Z[p']\, c$, increment p, and continue processing T from position i'. The process finishes once we get $c =$ "\$".*

The output of the LZ78 compressor is essentially the sequence of pairs (p', c) found at each step p of the algorithm. This parsing has a couple of important properties. First, all the phrases in an LZ78 parsing are different from each other. Second, the prefixes of a phrase are phrases.

Fig. 28 shows the LZ78 parsing of our example text. The figure also illustrates the *Ziv-Lempel trie* $LZTrie$, which is the trie storing the set of strings Z. The trie has n' nodes (as there is one per string in Z). If $Z[p] = Z[p']\,c$, then node p is a child of p' by edge labeled c. $LZTrie$ is used in LZ78 compression to produce the Ziv-Lempel parse in $O(n)$ time. The other trie, $RevTrie$, is used by some indexes we review soon. Depending on the case, we can associate some extra information to these tries, such as the text position of the phrases, the lexicographical rank of the phrases (shown in the figure), and so on.

Definition 18 *The* Ziv-Lempel trie *of T is a trie storing all the phrases of the LZ78 parsing of T. Each node corresponds to a distinct phrase.*

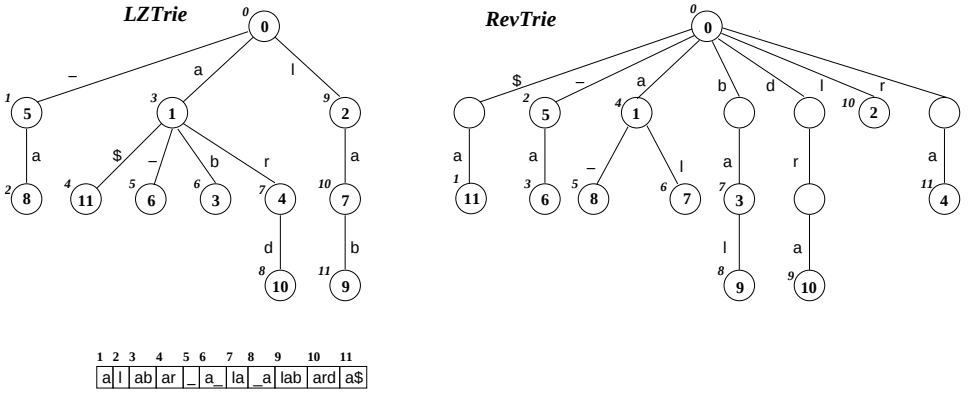

Fig. 28. On the bottom, the LZ78 parsing of the text `"alabar a la alabarda$"`. On the left we show $LZTrie$, the trie of the phrases $Z[p]$, and on the right $RevTrie$, the trie indexing the reverse phrases. We use the phrase identifiers as node labels. The smaller numbers in italics outside the nodes are the lexicographic rank of the corresponding strings.

An important property of both Ziv-Lempel parsings is that the number of phrases they produce is at most $n/\log_\sigma n$, and in general the size of the Ziv-Lempel compressed text (slowly) converges to the entropy of the source [Cover and Thomas 1991]. Of more direct relevance to us, it has been shown that n' is related to the definition of $H_k(T)$ we use in this paper [Kosaraju and Manzini 1999; Ferragina and Manzini 2005].

Lemma 7 *Let n' be the number of phrases produced by Ziv-Lempel parsing of text $T_{1,n}$, using either the parsing of Definition 16 or 17. 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 7 implies that the Ziv-Lempel trie can be stored, even using pointers, in $O(nH_k)$ bits of space. The pioneer work in Ziv-Lempel based indexes is due to Kärkkäinen and Ukkonen [1996a], deriving from their earlier work on sparse suffix trees [Kärkkäinen and Ukkonen 1996b]. This Ziv-Lempel based index is also the first compressed index we know of.

Before entering into the Ziv-Lempel based methods, let us review the sparse suffix tree [Kärkkäinen and Ukkonen 1996b], which is the first succinct index we know of. This 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 $\Sigma^i P$, $0 \leq i < h$. Overall this requires $O(\sigma^{h-1}m + occ)$ time. By choosing $h = 1 + \epsilon \log_\sigma n$ we get $O(mn^\epsilon + occ)$ search time and $O((n \log n)/h) = O(n \log \sigma)$ bits of space.

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 Ziv-Lempel parsing of T. This parsing is a variant of the original Ziv-Lempel parsing (Definition 16). Although they use the property that $n' \leq n/\log_\sigma n$ to show that their index is succinct, requiring $O(n \log \sigma)$ bits of space, Lemma 7 shows that they actually require $O(nH_k)$ bits of space. We present here the results in their definitive form [Kärkkäinen 1999].

The original 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). 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 $occ = occ_p + occ_s$.

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 avoid reporting the same occurrence multiple times, they insist that $P_{1,i}$ has to be completely included in a phrase, so the partitioning $(P_{1,i}, P_{i+1,m})$ will be unique per occurrence. The occurrences of $P_{i+1,m}$ are found using a sparse suffix tree that indexes the phrase beginnings (and is searched only for phrase-aligned occurrences). Those of $P_{1,i}$ within a phrase are found using the equivalent of $RevTrie$, which is searched for $P_i P_{i-1} \ldots P_1$ ($RevTrie$ is illustrated in Fig. 28 for the LZ78 parsing). For example, $P =$`"labar"` appears twice in Fig 28: in phrase 2 with partition (`"l"`,`"abar"`) and in phrase 9 with partition (`"lab"`,`"ar"`). For succinctness we will use the LZ78 parsing of Fig. 28 to illustrate the method, although it actually uses the original Ziv-Lempel parsing of Definition 16.

Each of those two searches yields a *lexicographical range* in $[1, n']$. The sparse suffix tree yields the range $[l_2, r_2]$ of the phrase-aligned suffixes that start with $P_{i+1,m}$. The trie of reverse phrases 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 $(x_p, y_p) \in [l_1, r_1] \times [l_2, r_2]$. This is a two-dimensional

range search problem (Section 6.4), where we store all the 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 $i = 3$, we find range $[l_1, r_1] = [8, 8]$ for "bal" in *RevTrie* (as node with label $p = 9$ is 8th in a preorder traversal of *RevTrie*; this data can be stored in each node), and range $[l_2, r_2] = [7, 8]$ for "ar" in the sparse suffix tree (as the 7th and 8th suffixes starting phrases start with "ar"). Then we search for $[8, 8] \times [7, 8]$ and find 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 original parsing, it can be ensured that no source contains another and thus source intervals can be linearly ordered. An array $S[p]$ of phrase numbers sorted by their source interval position in T, plus a bit array $B[1, n]$ indicating which text positions start phrase sources, permits finding each phrase copying $[j, j + m - 1]$ in constant time: $S[rank_1(B, j)]$ is the last phrase in S whose source starts in $T_{1,j}$. We traverse S backwards from that position until the source intervals finish before T_{j+m-1}.

The index space is $O(n' \log n) = O(nH_k)$ for the sparse suffix tree and the trie of reverse phrases. Among the range search data structures considered by Kärkkäinen [1999], we take those requiring $O(\frac{1}{\varepsilon} n' \log n')$ bits of space (the one we reviewed in Section 6.4 corresponds to $\epsilon = 1$). Array S also needs the same space, and bit array B requires $O(n' \log n)$ bits using the techniques of Section 6.2. Thus the overall space is $O(\frac{1}{\varepsilon} nH_k)$ bits, in addition to the text.

We note that this index carries out counting and locating simultaneously. The $m - 1$ searches in *RevTrie* (for all $P_{1,i}$ reversed) and sparse suffix tree (for all $P_{i+1,m}$) require $O(m^2)$ time. Finding the secondary occurrences takes constant time per occurrence retrieved, so it amounts to $O(occ_s)$. The remaining cost is that of the two-dimensional search. Depending on the structures used, different tradeoffs are obtained [Kärkkäinen 1999]. We give the most interesting one for us.

Theorem 22 (Kärkkäinen and Ukkonen [1996a]) *The* LZ-Index *(*KU-LZI*) offers the following space/time tradeoffs.*

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	$k = o(\log_\sigma n)$*;*
	$0 < \epsilon < 1$ *(not necessarily constant)*

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. Finally, we point out a variant of this idea [Kärkkäinen and Sutinen 1998] that can answer queries in $O(m)$ counting time and $O(1)$ locating time per occurrence, for short patterns ($m < \log_\sigma n$).

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 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"`. Note that position t in T, belonging to the p-th phrase, corresponds to position $rev(t, p) = (n - t + 1) + (n' - p + 1)$ in T^R. 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 Ziv-Lempel trie of T; (4) *Range*, a two-dimensional range search data structure similar to that of the KU-LZI. The first three structures require $O(nH_k)$ bits of space, yet *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 the *LZTrie*). For example, in Fig. 28, 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 *pioneer* for P if P is a suffix of $Z[p]$. In our example the pioneer nodes for $P =$`"a"` are 1, 7, and 8. Then, all secondary occurrences correspond to *LZTrie* nodes that descend from pioneer nodes (including themselves). To find the secondary occurrences, it is sufficient to obtain the pioneer nodes and then traverse all their subtrees reporting all the text positions found (with the appropriate offsets).

Finding the pioneer nodes is easy using the FMI of T^R. It is a matter of searching for P^R, as that corresponds to occurrences of P that are phrase suffixes, or which is the same, to occurrences of $P\#$ in $T^\#$. 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. 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 S of pointers to the corresponding *LZTrie* nodes. As the range for P^R is always contained in the area covered by S, vector S permits finding the pioneer nodes of the results of the search. As S has n' entries, it occupies $nH_k + o(n \log \sigma)$ bits. Thus the occ_s secondary occurrences are reported in $O(m+occ_s)$ time. Fig. 29 illustrates.

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

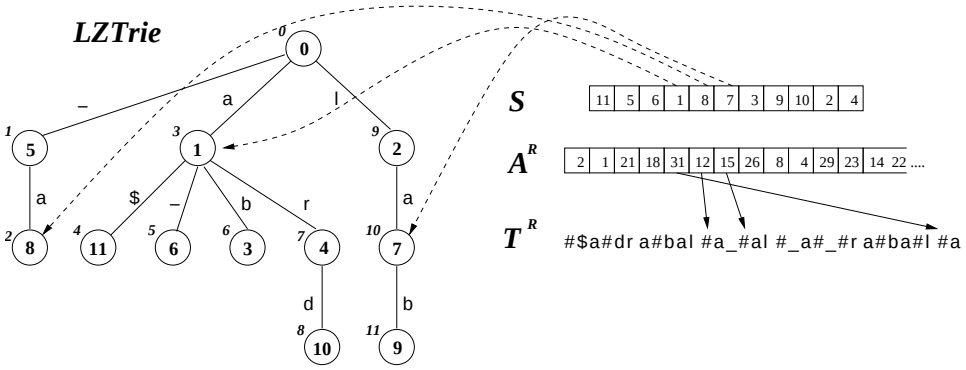

Fig. 29. Parts of the FM-LZI index. We show T^R and A^R (none of which is explicitly represented), as well as vector S and $LZTrie$. Only the part of A^R pointing to suffixes starting with "#" is shown in detail. This is the part S is aligned to. For legibility, S shows phrase numbers instead of pointers to $LZTrie$. The result of the search for `"#a"` is illustrated with the actual pointers from A^R to T^R and from S to $LZTrie$.

search gives us all the ranges $[sp_{i+1}, ep_{i+1}]$ in A corresponding to the occurrences of $P_{i+1,m}$, $1 \leq i < m$ (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 $[sp^R_i, ep^R_i]$ of the occurrences of $\#P_i P_{i-1} \ldots P_1$ in A^R, $1 \leq i < m$. 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 9. Using the FMI of T^R we get $[sp^R_3, ep^R_3] = [8, 9]$ (A^R range for `"#b"`, corresponding to `"b#"` in $T^{\#}$), and we get that $[sp^R_2, ep^R_2]$ (A^R range for `"#ab"`, corresponding to `"ba#"` in $T^{\#}$) is empty, see Fig. 29. 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 29, we see that $A^R[8] = 26$ in T^R matches with $A[12] = 5$ in T and $A^R[9] = 8$ in T^R matches with $A[13] = 17$ in T. For example, $t = 5$ in T matches with 26 in T^R because it belongs to the 4th phrase and thus t corresponds to $rev(5, 4) = (21 - 5 + 1) + (11 - 4 + 1) = 25$ in T^R.

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

They use the two-dimensional data structure [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 $1 \leq i < m$, 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 $O(n' \log n) = O(nH_k)$ bits. Note that the space of the structure, $O(n' \log^{1+\gamma} n')$ bits, is $O(nH_k \log^{\gamma} n) + o(n \log\sigma \log^{\gamma} n)$ 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 t that start a phrase in *Range*, we add all positions $[t - \log\log n + 1, t]$. 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 $o(n \log\sigma)$ if $\sigma = o(n^{1/\log\log n})$. For example, this is valid whenever $\sigma = O(n^\beta)$ for any $0 < \beta < 1$.

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 23 (Ferragina and Manzini [2005]) *The* LZ-Index *(*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 is due to other results by Alstrup et al. [2000], which need $O(n' \log n' \log\log n')$ bits of space and can search in time $O((\log\log n)^2 + res \log\log n)$. We can retain the same 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.

10.4 The LZ-Index of Navarro (NAV-LZI)

The *LZ-Index* of Navarro [2002, 2004] (NAV-LZI) uses essentially $LZTrie$ and $RevTrie$ (Definition 18), under LZ78 parsing. Despite that its complexity is not competitive, the structure is interesting because it is the only self-index not using the suffix array concept at all.

The NAV-LZI uses four structures: (1) $LZTrie$, the Ziv-Lempel trie of T, (2) $RevTrie$, storing the reverse of strings $Z[p]$, (3) the same two-dimensional $Range$ data structure of Section 10.2, (4) $Node$, a mapping from phrase numbers to $LZTrie$ nodes.

To save space, $LZTrie$ and $RevTrie$ tree shapes are represented using parentheses rather than full pointers. The tree shape of Fig. 28 is expressed by a preorder traversal as `"((())(()()()(()))(((()))))"`, or $Tr = 11100110101010110001110000$ in bits. Munro and Raman [1997] show how a tree of n' nodes can be represented using $2n' + o(n')$ bits so that it can be traversed in constant time per operation, by identifying each node with its open parenthesis. Some operations implemented in constant time are: first child, next sibling, parent, depth, preorder position, etc. For example, the first child of node at position t is $t+1$ unless $Tr[t+1] = 0$, in which case the node is a leaf. The depth of node at position t is $rank_1(Tr, t) - rank_0(Tr, t)$. Other operations are more complex to implement, but the ideas derive from those in Section 6.1.

In addition to the tree space, $LZTrie$ has to store the characters labeling the edges and the phrase number of each node. These are stored in arrays, in preorder traversal. In our example, the characters array is $lets[1, 12] =$ `"ε aa$ brdlab"` and the phrase numbers array is $ids[1, 12] = \{0, 5, 8, 1, 11, 6, 3, 4, 10, 2, 7, 9\}$. Let $pos(v)$ be the preorder position of node v, $pos(v) = rank_1(Tr, t)$, where t represents node v. For example, the node v representing string $S^v =$`"ar"` corresponds to the open parenthesis at $Tr[13]$. It is the 8th tree node in preorder traversal, $pos(v) = rank_1(Tr, 13) = 8$. With position 8, we know that node v descends by character $lets[8] =$ `"r"` from its parent, and that its phrase number is $ids[8] = 4$ (see Fig. 28). Note also that, if v is represented by $Tr[t]$ and t' is the closing parenthesis that matches t, then $[rank_1(Tr, t), rank_1(Tr, t')]$ is the interval of positions of all nodes that descend from v in the trie. For our example node, this interval is $[rank_1(Tr, 13), rank_1(Tr, 16)] = [8, 9]$, which contains v itself as well as the node representing phrase 10, `"ard"`.

$RevTrie$ is a bit more complicated. Since there are $RevTrie$ nodes not corresponding to any phrase (see Fig. 28), there could be more than n' nodes in $RevTrie$. To limit its size, we compress unary paths as when moving from suffix tries to suffix trees (Section 3.2). This ensures having at most $2n'$ nodes in $RevTrie$. Clark and Munro [1996] show how a suffix tree can be represented with parentheses plus some additional information regarding the first character of each edge label and its length (like a Patricia tree [Morrison 1968]). The NAV-LZI is more extreme, as nothing apart from the tree shape (using parentheses) and the array of phrase numbers is

stored. Let p be the phrase number of node v^R in $RevTrie$, representing string S^{v^R}. Then $v = Node(p)$ is the corresponding $LZTrie$ node for phrase p. Using the parent traversal operation in $LZTrie$ one can obtain S^v in reverse order, and this is precisely S^{v^R}. This is the information we need to traverse $RevTrie$: In order to descend from $RevTrie$ node v^R to the child $(v^R)'$ labeled by some given character c, we need to obtain the full string of each child, character by character, at overall cost $O(\sigma|S^{v^R}|)$. Thus, finding the $RevTrie$ node corresponding to a string S costs $O(|S|^2\sigma)$ time, whereas the same search on $LZTrie$ costs only $O(|S|\sigma)$ time. By converting the alphabet to binary, the $O(\sigma)$ terms become $O(\log\sigma)$ (we omit the technicalities).

Occurrences are classified according to how many phrases they span. Occurrences of type 1 are contained in a phrase, those of type 2 are contained in a pair of consecutive phrases, and those of type 3 span three phrases or more. Thus we have $occ = occ_1 + occ_2 + occ_3$ occurrences to count and locate.

Occurrences of type 1 are found with the same idea for secondary occurrences in Section 10.3, yet with a different mechanism. We search for P reversed in $RevTrie$. Say we arrive at node v. Any node v' descending from v, including $v' = v$ itself, corresponds to a phrase terminated with P. Thus $Node(v')$ is a pioneer $LZTrie$ node. Thus we traverse and report all subtrees of all those $Node(v')$ nodes in $LZTrie$, so occurrences of type 1 are counted and located in $O(m^2\log\sigma + occ_1)$ time.

Occurrences of type 2 are found with the range search data structure, as in Section 10.3. This time we search for $P_{1,i}$ reversed in $RevTrie$, and for $P_{i+1,m}$ in $LZTrie$, obtaining two nodes v_{rev} and v_{lz}, respectively. We are interested in phrases p such that the node of phrase p descends from v_{rev} and that of phrase $p+1$ descends from v_{lz}. We store in $Range$ all points $(pos(v'), pos(v))$ for each phrase $1 \le p < n'$, so that $v' \in RevTrie$ has phrase number p and $v \in LZTrie$ has phrase number $p+1$. Thus a range search for the array areas that descend from v_{rev} and v_{lz} retrieves those occurrences we want. To implement $Range$, the two-dimensional data structure of Section 6.4 is used to store n' points in a grid $[1, n'] \times [1, n']$ in $n'\log n'(1 + o(1))$ bits, and can find the res results of a query in $O((1 + res)\log n')$ time. Thus occurrences of type 2 cost $O(m^3\log\sigma + (m + occ_2)\log n)$ time.

Finally, for occurrences of type 3, the fact that every LZ78 phrase is unique is exploited. Since an occurrence of type 3 must contain a whole phrase, there cannot be more than $m(m-1)/2 \ge occ_3$ such occurrences because this is the number of substrings of P. Essentially, the phrase equal to each substring of P is found using $LZTrie$ and extended to a full occurrence if possible, in overall time $O(m^2\log\sigma + m^3)$.

Overall, the query time is upper bounded by $O(m^3\log\sigma + (m + occ)\log n)$. The space is dominated by the phrase arrays of $LZTrie$ and $RevTrie$, the $Range$ structure, and the $Node$ mapping. Each of these requires $n'\log n'(1 + o(1))$ bits, which makes the index space $4nH_k + o(n\log\sigma)$ bits if $k = o(\log_\sigma n)$.

We note that in this index counting is not easier than locating. Just like the KU-LZI, both processes must be carried out simultaneously. In order to display a text substring, we first locate its phrase p and then traverse $LZtrie$ upwards from $Node(p)$ to obtain all its characters in reverse order. If we need more phrases, we

repeat the process with $p-1$ or $p+1$, and so on. This takes $O(\log \sigma)$ time per character, yet it displays whole phrases. The extra work to display partial phrases is $O(n/n') = O(\log(n)/H_k)$ on average.

Theorem 24 (Navarro [2004]) *The* LZ-Index *(*Nav-LZI*) offers the following space/time tradeoff.*

Space in bits	$4nH_k + o(n \log \sigma)$
Time to count	$O(m^3 \log \sigma + (m + occ) \log n)$
Time to locate	*free after counting*
Time to display ℓ chars	$O(\log \sigma(\ell + \log(n)/H_k))$ *(last term is on average)*
Conditions	$k = o(\log_\sigma n)$

Note that, by using the range search data structure of Section 10.3 [Alstrup et al. 2000], the space would be $O(nH_k \log^\gamma n)$ and the counting time would decrease to $O(m^3 \log \sigma + m \log \log n + occ)$.

We also remark that, very recently, Arroyuelo et al. [2006] presented a variant of this index which reduces the space complexity to $(2+\epsilon)nH_k + o(n \log \sigma)$ for any $\epsilon > 0$, and the search complexity to $O(m^2 \log m + (m + occ) \log n)$. The displaying complexity also improves: It becomes $O(\ell / \log_\sigma n)$ worst case by using recent techniques [Sadakane and Grossi 2006].

In practice: The implementation of the Nav-LZI has several differences with respect to the theoretical proposal. The most important one is that *Range* structure is replaced by $RNode$, a mapping from phrase numbers to $RevTrie$ nodes. Now occurrences of type 2 are found as follows: Check each phrase p in the range descending from v_{rev} and report it if phrase $p+1$ descends from v_{lz}. Each such check takes constant time with $Node$. Yet, if the range of v_{lz} has less elements, do the opposite: check phrases from v_{lz} in v_{rev}, using $RNode$. $RNode$ is also useful to reduce the usage of the slower $RevTrie$ to a minimum: In every case where we can search for the reverse string in $LZTrie$, we avoid searching $RevTrie$ as we can map results using $RNode$.

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. Just as an overview, we provide a summary in Table 1.

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 Ziv-Lempel based methods, as they are totally different. For this discussion, the table of Fig. 19 is particularly enlightening. Each number $i \in [1, n]$ can be classified according to two parameters: the list (or table column)

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. We refer to the theorems given earlier for accurate statements and boundary conditions.

Index	entropy term	time to count	Theorem	page
MAK-CSA	$2nH_k \log n$	$O(m \log n + \text{polylog}(n))$	11	34
GV-CSA	nH_0	$O(m \log^2 n)$	12	37
SAD-CSA	nH_0	$O(m \log n)$	13	40
MN-CCSA	$nH_k \log n$	$O(m \log n)$	14	42
GGV-CSA	nH_k	$O(m \log \sigma + \text{polylog}(n))$	16	47
FMI	$5nH_k$	$O(m)$	17	52
HUFF-FMI	$2nH_0$	$O(mH_0)$	18	53
WT-FMI	nH_0	$O(m)$	19	55
RL-FMI	$nH_k \log \sigma$	$O(m)$	20	58
AF-FMI	nH_k	$O(m)$	21	60
KU-LZI	$O(nH_k)$	$O(m^2 + (m + occ) \log n)$	22	63
FM-LZI	$O(nH_k \log^\gamma n)$	$O(m)$	23	66
NAV-LZI	$4nH_k$	$O(m^3 \log \sigma + (m + occ) \log n)$	24	69

$T_{A[i]-1}$ and the context (or table row) $T_{A[i],A[i]+k-1}$ where i belongs. Within a given row and column, numbers i (sharing $T_{A[i]-1,A[i]+k-1}$), are classified according to $T_{A[i]+k,n}$.

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

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

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 the wavelet tree as a device to represent to which column each row element belongs.

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 $T^{bwt}_i = T_{A[i]-1}$, that is, the column identifiers of the positions i 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 $A[i]$ and use the normal binary search on A, the FMI family uses a completely different form of searching, which directly relies on T^{bwt}.

The final practical twist of the GGV-CSA is the discovery that γ- or δ-encoding of consecutive i 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.

A completely different mechanism to achieve $O(nH_k)$ space is used in the other indexes. These rely on compressing the runs that appear in Ψ, or alternatively those that appear in T^{bwt}. The former 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 an FMI 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 the results of Grossi et al. [2004]. 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 Ziv-Lempel methods in this discussion and in the table.

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

	local entropy	run lengths
using Ψ	SAD-CSA (nH_k, $m \log n$) GGV-CSA (nH_k, $m \log \sigma$)	MAK-CSA ($2nH_k \log n$, $m \log n$) 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 approaches 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 at the same time works as a versatile* index *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, 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, the theory is not yet complete, as there are several open questions to be answered. A first open question is whether one can obtain nH_k space and $O(m)$ query time on any alphabet size, and in general which is the lower bound relating these parameters. One easily notices that obtaining nH_k without any constraints is impossible as $nH_k = 0$ for large k (in particular for $k = n$). Another example is a text consisting of a permutation of $[1, n]$, where $\sigma = n$. Although $H_k = 0$ for $k \geq 1$, $\Omega(n \log n)$ bits are necessary to represent such text. Still, there is a gap between what is obtained and what is possible. This statement also applies to the sublinear terms in the space complexities, which might be bigger than the entropy-related part. However, recent lower bounds [Miltersen 2005] on *rank* and *select* dictionaries seem to indicate that not much more progress in that direction is possible.

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 small alphabets, using $O(nH_k \log^{\gamma} n)$ 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 the exact search problem in main memory. There are many more further challenges, with regard to more complex searching, index construction and updating issues, 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 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]. This is not, however, a succinct structure. 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 by Kärkkäinen and Rao [2003].

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]. There is also some recent work on efficient construction of (plain) suffix arrays [Manzini

and Ferragina 2004] and trees [Farach 1997]. 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. 1998; Clifford and Sergot 2003]. For further details on the topic, see [Aluru 2005, Chapter 5].

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. Some recent work on compressed indexes can be found [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. 2004; Lam et al. 2005]. 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 utmost 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 to become aware of the intriguing opportunities provided by these new techniques, for a successful technology transfer to take place. The community is working on this: To faciliate the chance for smooth transfer from prototype implementations to real use, a repository of standardized library implementations has been made available at mirrors `pizzachili.dcc.uchile.cl` and `pizzachili.di.unipi.it`. There have been articles about the FM-Index published 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 papers on the topic can be found in the latest *Workshop on Efficient and Experimental Algorithms (WEA)* conferences.

Overall, we believe that self-indexing is one of the most exciting research directions within 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 Kunihiko Sadakane for pointing out some mistakes in the original version of this paper, as well as the anonymous reviewers for improving its readability.

REFERENCES

Abouelhoda, M., Ohlebusch, E., and Kurtz, S. 2002. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS v. 2476 (2002), pp. 31–43.

Alstrup, S., Brodal, G., and Rahue, 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.

Arroyuelo, D., Navarro, G., and Sadakane, K. 2006. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS (2006). To appear.

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.

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. and Sergot, M. 2003. Distributed and paged suffix trees for large genetic databases. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 2676 (2003), pp. 70–82.

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. 1998. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS)* (1998), pp. 174–185.

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. 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. 2004. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2004), pp. 655–663.

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. 2004a. 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. 2004b. Succinct representation of sequences. Technical Report TR/DCC-2004-5 (Aug.), Department of Computer Science, University of Chile, Chile.

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.

Giegerich, R., Kurtz, S., and Stoye, J. 1999. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE)*, LNCS v. 1668 (1999), pp. 30–42.

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. Short paper. Full version as Technical Report TR/DCC-2004-4, Department of Computer Science, University of Chile, July 2004.

Grabowski, S., Mäkinen, V., Navarro, G., and Salinger, A. 2005. A simple alphabet-independent fm-index. In *Proceedings of the 10th Prague Stringology Conference (PSC'05)* (2005), pp. 230–244.

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.

Huffman, D. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E. 40*, 9, 1090–1101.

Huynh, T., Hon, W.-K., Lami, T.-W., and Sung, W.-K. 2004. Approximate string matching using compressed suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 3109 (2004), pp. 434–444.

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. 2003. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 2676 (2003), pp. 186–199.

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

Ko, P. and Aluru, S. 2003. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS v. 2676 (2003), pp. 200–210.

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 (2006). To appear.

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. 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.

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 Grossi, R. 2006. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2006), pp. 1230–1239.

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* (second ed.). Morgan Kaufmann Publishers.

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3, 337–343.

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.