Università di Pisa

Dipartimento di Informatica

Technical Report : TR-00-03

Opportunistic data structures with applications

Paolo Ferragina, Giovanni Manzini

March 30, 2000

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Opportunistic Data Structures with Applications

P. Ferragina* G. Manzini†

March 30, 2000

Abstract

There is an upsurging interest in designing succinct data structures for basic searching problems (see [32] and references therein). The motivation has to be found in the exponential increase of electronic data nowadays available which is even surpassing the significant increase in memory and disk storage capacities of current computers. Space reduction is an attractive issue because it is also intimately related to performance improvements as noted by several authors (e.g. Knuth [20], Bentley [6]). In designing these *implicit* data structures the goal is to reduce as much as possible the auxiliary information kept together with the input data without introducing a significant slowdown in the final query performance. Yet input data are represented in their entirety thus taking no advantage of possible repetitiveness into them. The importance of those issues is well known to programmers who typically use various tricks to squeeze data as much as possible and still achieve good query performance. Their approaches, thought, boil down to heuristics whose effectiveness is witnessed only by experimentation.

In this paper, we address the issue of compressing and indexing data by studying it in a theoretical framework. We devise a novel data structure for indexing and searching whose space occupancy is a function of the entropy of the underlying data set. The novelty resides in the careful combination of a compression algorithm, proposed by Burrows-Wheeler [8], with the structural properties of a well known indexing tool, the Suffix Array [23]. We call the data structure *opportunistic* since its space occupancy is decreased when the input is compressible at no significant slowdown in the query performance and without any assumption on a particular fixed distribution. More precisely, its space occupancy is optimal in a *information-content* sense because a text $T[1,u]$ is stored using $O(H_k^*(T)) + o(1)$ bits per input symbol, where $H_k^*(T)$ is the kth order entropy of T (the bound holds for any fixed k). Given an arbitrary string $P[1,p]$, the opportunistic data structure allows to search for the occ occurrences of P in T requiring $O(p + occ \log^\epsilon u)$ time complexity (for any fixed $\epsilon > 0$). If data are non compressible, then we achieve the best space bound currently known [14]; otherwise our solution improves the succinct suffix array in [14] and the classical suffix tree and suffix array data structures either in space or in query time complexity or both.

It was a belief [39] that some space overhead should be paid to use full-text indices (i.e. suffix trees or suffix arrays) with respect to the word-based indices (i.e. inverted lists). The results in this paper show that *no space overhead* is needed at all, and as an application we improve space and query time complexity of the well-known Glimpse tool [24].

We finally investigate the modifiability of our opportunistic data structure by studying how to choreograph its basic ideas with a dynamic setting thus achieving effective searching and updating time bounds.

*Dipartimento di Informatica, Università di Pisa, Italy.

†Dipartimento di Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, Alessandria, Italy and IMC-CNR, Pisa, Italy.

1 Introduction

Data structure is a central concept in algorithmics, and computer science in general. In the last decades, it has been investigated from different points of view and its basic ideas enriched by new functionalities with the aim to cope with the features of the peculiar setting of use: dynamic, persistent, self-adjusting, implicit, fault-tolerant, just to cite a few. Space reduction in data structural design is an attractive issue, now more than ever before, because of the exponential increase of electronic data nowadays available, and because of its intimate relation with algorithmic performance improvements (see e.g. Knuth [20] and Bentley [6]). This has recently motivated an upsurging interest in the design of succinct, *implicit* data structures for basic searching problems (see [32] and references therein). The goal in their design is to reduce as much as possible the auxiliary information kept together with the input data without introducing any significant slowdown in the final query performance. Input data are nonetheless represented in their entirety thus taking no advantage of possible repetitiveness into them. The importance of those issues is well known to programmers who typically use various tricks to squeeze data as much as possible and still achieve good query performance. Their approaches, thought, boil down to heuristics whose effectiveness is witnessed only by experimentation.

In this paper we address the issue of compressing and indexing data by studying it in a theoretical framework. We devise a novel data structure for indexing and searching whose space occupancy is a *function of the entropy* of the underlying data set. The data structure is called *opportunistic* in that, although no assumption on a particular fixed distribution is made, it takes advantage of compressibility of the input data by decreasing the space occupancy at *no significant slowdown* in the query performance. The resulting data structure is provably space optimal in a stronger *information-content* sense because it stores a text $T[1, u]$ using $O(H_k^*(T)) + o(1)$ bits per input symbol (for any fixed $k \geq 0$), where $H_k^*(T)$ is the kth order entropy of T. $H_k^*(T)$ is the maximum compression achievable using for each character a code that only depends on the k characters preceding it and with the additional requirement that the coding of any string s takes at least $\log |s|$ bits (for details see Appendix A. From the best of our knowledge no other result is known in the literature about the study of the interplay between compression and indexing of data collections. The exploitation of data compressibility have been already investigated only with respect to its impact on algorithmic performance in the context of on-line algorithms (e.g. caching and prefetching [21, 19]), string-matching algorithms (see e.g. [1, 2, 12]), sorting and computational geometry algorithms [9].

The Scenario. The study, the design and the experimentation of methods for searching and updating text collections have attracted the attention of the algorithmic and data structural community during the last four decades. Numerous and precious ideas have been presented in the main literature — inverted lists, Patricia trees, tries, suffix trees, suffix arrays, just to cite a few — and these constitute the *heart* of a huge amount of software tools currently in use for the processing of textual data. The main idea underlying these data structures is to build an *index* that allows to focus the search for a given queried pattern (string) only on a very small portion of the available text. The improvement in the query performance is paid by the additional space necessary to store the index. Most of the research in this field has been therefore directed to design data structures which offer a good trade-off between query and update time versus the space usage. There are two main approaches: *word-based* indexes and *full-text* indexes. Word-indexes exploit the fact that for natural linguistic texts, the universe of distinct words is small, and thus store all of these occurrences in a trie or a table; so they offer very fast word (or prefix-word) queries and occupy reduced space (about 20-30% of the original text size [39]), but they are unable to index *unstructured texts* (e.g. DNA-sequences or variable-length records). Full-text indexes have been then designed to overcome these limitations at the cost of an increase in the additional space occupied by the underlying index. Examples of such indexes are: suffix trees [27], suffix arrays [23] and String B-trees [13]. Despite their superior features (elegance, versatility, and guaranteed performances), they have not seen widespread use in string processing software mainly because, although asymptotically optimal in space, the constants hidden in the big-Oh notation range between 4 and 20 [22]. Such a value is sometimes a significant bottleneck that may even prevent the use of those data structures in practical applications. Some progress has been recently achieved [32, 14] but nonetheless an asymptotical linear space *seems* unavoidable and this makes inverted lists (and their derivatives) much more appealing when space issue is a primary concern. It is in fact a current belief [39, 4] that some space overhead must be paid to use full-text indices with respect to the word-based indices.

In this context compression appears always as an attractive choice, if not mandatory. Processing speed is currently improving at a faster rate than disk speeds, and since compression decreases the demand of storage devices at the expenses of processing, it is becoming more economical to store data in a compressed form than uncompressed. It goes without saying that compression may introduce some improvements which are surprisingly not confined to the space occupancy. Transfer time and, even, seek times can benefit from the space reduction induced by the text compression [20, 6]. In extreme cases, compression might mean that one less level of indexing blocks is necessary in systems such as UNIX, in which files are stored as a tree of blocks.

Previous Results. Starting from these promising considerations, many authors have recently tried to combine text compression with algorithms and indexing techniques. Most of the research has been mainly concentrated on the *compressed matching problem*, introduced in [1], as the task of performing string matching in a compressed text without decompressing it. A collection of algorithms is currently known to solve efficiently (possibly optimally) this problem on text compressed by means of various schemes: e.g. run-length [1], LZ77 [12], LZ78 [2], Anti-dictionaries [10]. All of those results, however, rely on a full scan of the whole *compressed* text. So that, although asymptotically faster than the classical scan-based methods [11], which operate on the uncompressed texts, their overall time requirement is still unacceptable when they work on large text collections.

Approaches to combine compression and indexing techniques are nowadays receiving more and more attention, especially in the context of word-based indices, by achieving experimental trade-offs between space occupancy and query performance (see e.g. Glimpse [24, 39, 5, 4]). The only notable idea towards the direct compression of the index data structure, has been proposed in [17, 18] where the properties of the Lempel-Ziv's compression scheme have been exploited to reduce the number of *index points*, still supporting pattern searches. As a result, the overall index requires provably sublinear space but at the cost of either limiting the search to q-grams [17] or worsening the query performance to be possibly superlinear in the pattern length [18].

A natural question arises at this point: Do full-text indexes need a space occupancy *linear* in the (uncompressed) text size u in order to support searches for arbitrary patterns $P[1, p]$ in (worst-case) time depending only on p and as much independent as possible of u ? (See [14]'s conclusion section). It is a common belief [39, 4] that some space overhead must be paid to use the full-text indices, but is this actually a provable need ?

Our Results. In this paper we answer the two questions above by providing the so called *opportunistic data structure*[1]. Although no assumption on any particular fixed distribution is made, the space occupancy of the opportunistic data structure is provably decreased when the

[1]The concept of *opportunistic algorithm* has been introduced in [12] to characterize an algorithm which takes advantage of the compressibility of the text to speed up its (scan based) search operations. In our paper, we turn this concept into the one of *opportunistic data structure*.

input is compressible at no significant slowdown in the query performance. The data structure is space optimal in a stronger *information-content* sense because a text $T[1,u]$ can be stored using $O(H_k^*(T)) + o(1)$ bits per input symbol, where $H_k^*(T)$ is the kth order entropy of T (the bound holds for any fixed k). Given an arbitrary string $P[1,p]$, the opportunistic data structure allows to search for the *occ* occurrences of P in T requiring $O(p + occ \log^\epsilon u)$ time, for any fixed $\epsilon > 0$.

The novelty of our approach resides in the careful combination of the compression algorithm proposed by Burrows-Wheeler [8] with the the Suffix Array data structure [23] to obtain a sort of *compressed* suffix array. We indeed show how to *augment* the information kept by the BW-algorithm for decompressing purposes, in order to support effective *random accesses* to the compressed data without the need of uncompressing all of them at querying time. We design two algorithms for operating on our opportunistic data structure: one constitutes a novel effective approach to search for an arbitrary pattern $P[1,p]$ in a *compressed* suffix array taking $O(p)$ time in the worst case (see Section 4.1); the other algorithm exploits compression to fasten the retrieval of the actual positions of the pattern occurrences thus incurring only in a sublogarithmic slowdown and still acting on the compressed data structure (see Section 4.2).

In some sense, our result can be interpreted as a method to *compress* the suffix array which is commonly considered uncompressible (by using standard compression algorithms only 5% saving has been achieved [29]), and still support effective searches for arbitrary patterns into it. In their seminal paper, Manber and Myers [23] introduced the suffix array data structure showing how to search for a pattern $P[1,p]$ in $O(p + \log u + occ)$ time in the worst case. The data structure used $O(u \log u)$ bits of storage. Recently, Grossi and Vitter [14] proposed a method to reduce the space usage of suffix arrays thus achieving $\Theta(u)$ bits of storage. Their compact suffix array can be accessed via the subroutine `lookup(i)` which allows in $O(\log^\epsilon u)$ time to retrieve its i-th suffix. Searching in their compact suffix array via the classical Manber-Myers' procedure takes $O(p+\log^{1+\epsilon} u+occ \log^\epsilon u)$ time. Our solution therefore improves [14] both in space and query time complexity. As far as the problem of counting the pattern occurrences is concerned, our solution improves the classical suffix tree and suffix array data structures, because they achieve $O(p)$ time complexity but occupy $\Omega(u \log u)$ bits of storage. Notice that our opportunistic data structure achieves $O(p)$ searching time complexity but without the use of the so called *lcp*-array [23]. In [14] other hybrid indexing data structures based on the combination of their succinct suffix array and various known techniques [17, 33] have been introduced achieving $O(\frac{p}{\log u} + occ \log^\epsilon u)$ query-time complexity but yet requiring $\Omega(u)$ bits of storage.

In Section 5, we also investigate the modifiability of our opportunistic data structure by studying how to choreograph its basic ideas with a dynamic setting thus achieving efficient searching and updating time bounds. We show that a dynamic text collection Δ of size u can be stored in $O(H_k^*(\Delta)) + o(1)$ bit per input symbol (for any fixed $k \geq 0$ and not very short texts), support insert operations on individual texts $T[1,t]$ in $O(t \log u)$ amortized time and delete operations on T in $O(t \log^2 u)$ amortized time, and search for a pattern $P[1,p]$ in $O(p \log^3 u + occ \log u)$ time in the worst case.

Finally, we investigate applications of our ideas to the development of novel text retrieval systems based on the concept of block addressing (first introduced in the Glimpse tool [24]). The notable feature of block addressing is that it can achieve both sublinear space overhead and sublinear query time, whereas inverted indexes pointing to words or documents achieve only the second goal [4]. Unfortunately, all the known block addressing indices [24, 34] achieve this goal under some restrictive conditions on the model parameters (i.e. the block size). We show how to use our opportunistic data structure to devise a novel block addressing scheme, called `CGlimpse` (standing for *Compressed Glimpse*), which always achieves time and space sublinearity.

2 Background and Notations

Let us given a text $T[1,u]$, a corresponding compressed string $Z[1,z]$, and a pattern $P[1,p]$, drawn from a constant-sized alphabet Σ. The *compressed pattern matching* problem [1] consists of finding all the positions in T where the pattern P occurs, using only P and Z. A naive algorithm which first decompress Z and then searches P in it via standard string-matching algorithms would require $O(u+p)$ time. An efficient algorithm should not decompress the text but operate as much as possible on its compressed version. Scan-based algorithms are known to search P directly in Z requiring $o(u+p)$ time, whenever $z = o(u)$ and $p = o(u)$ (see above). In this respect they have been called *opportunistic* [12] because they take advantage of the compressibility of the text to speed up the (scan based) search operations. In our paper, we are interested in the problem of *indexing* compressed texts so that we turn the concept of *opportunistic algorithm* into the one of *opportunistic data structure*; namely one that requires $\Theta(z) + o(u)$ bits of storage and still guarantees $O(p + occ \times polylog(u))$ query time in the worst case. In this context, the adjective *opportunistic* refers to the fact that the data structure takes advantage of the compressibility of the text to reduce its size (which turns out to be sublinear for low-entropy texts), at no significant penalty cost in the query performance with respect to the known uncompressed solutions (e.g. suffix trees, suffix arrays, *etc.*). Opportunistic data structures are asymptotically better than suffix trees and suffix arrays for checking the presence of the pattern in the compressed text. In general they provide a *pseudo-optimal solution* to the problem of indexing compressed texts; a *full optimality* would mean the removal of the polylogarithmic multiplicative term in the query bound, but this is still an open problem even in the uncompressed setting [14].

For the sake of presentation, let $T[1,i]$ denote the ith prefix of the text T and $T[i,u]$ denote the ith suffix of T. A central concept in our discussion will be the *suffix array* data structure [23]. The suffix array $\mathcal{A}$ built on the string $T[1,u]$ is an array containing the lexicographically ordered sequence of the suffixes of T, represented via pointers to their starting positions (i.e., *integers*). For instance, if $T = ababc$ then $\mathcal{A} = [1,3,2,4,5]$. This way $\mathcal{A}$ occupies $u\lceil \log_2 u \rceil$ bits (see [14] for the removal of the logarithmic term). Manber and Myers [23] introduced this data structure in the early 90s and proposed an interesting algorithm to efficiently search for an arbitrary string $P[1,p]$ in T by taking advantage of the information coded into $\mathcal{A}$. They proved that all the occ occurrences of P in T can be retrieved in $O(p + \log u + occ)$ time in the worst-case provided that some additional information is given other than $\mathcal{A}$. The simplicity of the search procedure, the small constants hidden in the big-Oh notation, and the reduced space occupancy (wrt suffix trees) are the most important characteristics that made the suffix array very appealing for practical applications. In practice $\mathcal{A}$ occupies $4u$ bytes of storage which summed up to the text space gives overall $5u$ bytes of space occupancy. This is actually a lot when indexing large text collections. Many researchers have therefore tried to compress the array $\mathcal{A}$ to beat the u-bit-storage barrier, by using standard compression algorithms (e.g. Gzip) but without any success (only 5% saving has been reported in [29]). Consequently, it has been a long standing belief that suffix arrays were uncompressible because of the “apparently random” permutation of the suffix pointers stored in $\mathcal{A}$. Recent results in the data compression field have instead opened the door to revolutionary ways to compress suffix arrays. These novel ideas will constitute the basic blocks of our solution, and thus will be commented in detail in the next section.

3 The Burrows-Wheeler transform

In 1994, Burrows and Wheeler [8] proposed a data compression algorithm that achieves compression within a percent or so of that achieved by statistical modeling techniques, but at speeds comparable to those of algorithms based on Lempel-Ziv's (see e.g. [8, 37]). The algorithm does not process the input sequentially, but instead examines a block of text at a time, as a single unit. Possibly a block can be as large as the entire text. On each block is applied a reversible transformation (usually called *BW-transform*, BWT from now on) which gives a new block that contains the same characters (and hence has the same length), but it is easier to compress by simple compression algorithms. The BWT tends to group together characters which occur adjacent to similar text substrings (called *contexts*). This way the probability to find two occurrences of the same character close to each other is high in the transformed block. And this nice property is exploited by locally-adaptive compression algorithms, such as move-to-front coding [7], in combination with statistical (i.e. Huffman or Arithmetic coders) or structured coding models to succinctly represent the transformed block. The key tool is therefore the reversible BWT, and this will be discussed deeply below.

The reversible BW-transform. We use the classical text $T[1, 11] = mississippi$ as a running example to illustrate how the BWT works. We assume that the block size equals the text length, so that the entire text is processed at once. We distinguish between a *forward* transformation, which produces the block to be compressed, and a *backward* transformation which gives back the original text from the transformed one. Perhaps the most surprising thing is that the transformed text preserves enough information to go back to its original version, and this can be done in time linear in its length. In detail, the forward BWT consists of three basic steps (we follow the description in [8]):

1. Append a special character # to the end of T; this character is smaller than any other text character.
2. Form a *conceptual* $(u + 1) \times (u + 1)$ matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\#$ sorted in lexicographic order.
3. Construct the transformed text L by taking the last column of $\mathcal{M}$. Notice that the first row of $\mathcal{M}$ equals $\#T$.

There is a strong apparent relation between the matrix $\mathcal{M}$ and the suffix array $\mathcal{A}$ of the string $T\#$. By appending the special character # to the end of T, we ensured that no suffix of T is a prefix of another suffix. Hence, when sorting the rows of the matrix $\mathcal{M}$ we are essentially sorting the suffixes of $T\#$; everything occurring after the symbol # is never compared (and thus the circulating feature of the rows does not come into play). Consequently, entry $\mathcal{A}[i]$ points to the suffix of $T\#$ occupying (a prefix of) the ith row of $\mathcal{M}$. See Figure 1 as a running example. The cost of performing the forward BWT is given by the cost of constructing the suffix array $\mathcal{A}$ (i.e. step 2 above), and this requires $O(u)$ space and either $O(u \log u)$ time [23] or $O(u)$ time [27] (the constants in the space bound vary).

Conversely, the cyclic shift of the rows of $\mathcal{M}$ is crucial to define the backward BWT. In particular three, easy to prove, observations are the key tools to design the backward step [8].

Fact 1 *The matrix $\mathcal{M}$ and the transformed text L constructed by the forward BWT satisfy the following properties:*

a. *Every column of $\mathcal{M}$ is a permutation of its last column L; the first column of $\mathcal{M}$, call it F, is obtained by lexicographically sorting the characters in L.*

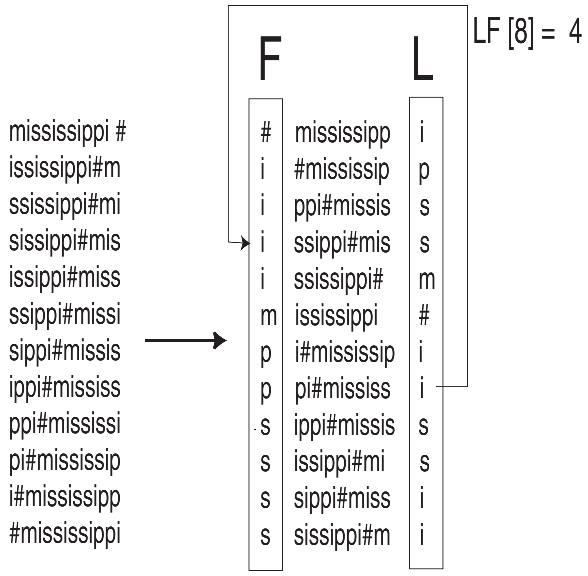

Figure 1: Original text $T = mississippi$, all cyclic shifts of the string $T\#$ on the right, and matrix $\mathcal{M}$ on the left. An entry of the array LF is shown in detail, namely $LF[8] = C['i'] + r_8 = 1 + 3 = 4$. Indeed, $L[8] =' i'$ is the third i in the prefix $L[1, 8]$, thus $r_8 = 3$. Consequently, the fourth row is the right cyclic shift of the eight row; and thus the character $L[8]$ corresponds to the character $F[4]$ (here we have L-to-F mapping).

b. *The lexicographic order between two rows of $\mathcal{M}$ ending with the same character remains the same even if we cyclically shift them toward the right (their last character becomes their first one, but they are equal).*

c. *Given the ith row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T, namely $T = \cdots L[i]F[i] \cdots$.*

Observation (c) indicates a way to find the character (i.e. $L[i]$) preceding a given one in T (i.e. $F[i]$), provided that the position (i.e. i) of its copy in F is known. This is not enough to reconstruct the whole text T, because we also need the step which maps a given character in L to its copy in F (it surely exists by observation (a)). This would complete the *cycle* and allow us to move again and again from one character in T to the previous one, via the strings L and F. Notice that T is reconstructed backward; reconstructing T forward is a simple exercise.

From the three observations above, we easily derive the following result which immediately gives the missing step.

Lemma 1 *Let $\mathcal{M}[i] = \alpha\, c$ be the ith row of the matrix $\mathcal{M}$, where α is a length-u string and c is a character of Σ, and let r_i be the rank of the row $\mathcal{M}[i]$ among all the rows ending with the character c. The row $c\,\alpha$, obtained via a cyclic right shift of $\mathcal{M}[i]$, has still rank r_i among the rows starting with c.*

Consequently, Lemma 1 gives a way to go from the occurrence of a character in L to the corresponding occurrence of the same character in F. We are therefore ready to describe the backward BWT (the correctness follows from Lemma 1):

1. Construct F by sorting lexicographically the characters in L.

2. Compute the array $C[1 \ldots |\Sigma|]$ indicating for each character c the number of times the characters in $\{\#, 1, \ldots, c-1\}$ occur in the original text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).

3. Define the array $LF[1 \ldots u+1]$ such that entry $LF[i]$ gives the position of the character $L[i]$ in the string F (i.e. LF stays for *L-to-F mapping*). Compute $LF[i]$ as $C[L[i]] + r_i$, where r_i equals the number of occurrences of character $L[i]$ in the prefix $L[1, i]$ (see also Lemma 1).

4. Reconstruct T backward as follows: set $s = 1$ and $T[u] = L[1]$ (because $\mathcal{M}[1] = \#T$); then, for each $i = u-1, \ldots, 1$ do $s = LF[s]$ and $T[i] = L[s]$. (Notice that s points to the position in F of the character $T[i+1]$; initially $s = 1$ points to $T[u]$.)

An illustrative example of the mapping LF is given in Figure 1. In [8] is shown how to compute all the ranks of the matrix rows in linear time and thus perform the overall backward BWT in $O(u)$ time and $O(u + |\Sigma|) = O(u)$ space.

It should result also clear from the discussion above how the suffix array $\mathcal{A}$ of the string $T\#$ can be derived from L: in Step 4, set $\mathcal{A}[s] = i+1$ after the updating of the variable s. Consequently, from the string L we can derive the suffix array of the original text T in linear time. This observation presented in [36] allowed to make the first significant step toward the compression of the Suffix Array data structure. However this result requires the *unpacking* of the string L, and thus the uncompression of the entire string Z, to explicitly construct the whole $\mathcal{A}$ and thus support fast searches on T. In this respect, the solution is not better than decompressing the text and then searching it via classical string-matching algorithms [11] or using opportunistic scan-based searching tools [12].

The discussion above has pointed out many properties of the string L, so that it is a pity not to be able to exploit its strong structure for avoiding its explicit overall unpacking. The *implicit* presence of the suffix array $\mathcal{A}$ into L does not give any advantage over the scan-based solutions if we need to pull it out before searching. It would be nice to keep string L compressed, and still be able to perform a binary search on the implicit array $\mathcal{A}$. This would allow us to take full advantage of the structure of L for fast searching and of its succinct encoding for reducing the overall space occupancy. This is actually the ultimate hope of any indexer: succinct and fast !

As we will show in the next section, this result may be achieved provided that a *sublogarithmic* slowdown is introduced in the cost of listing all pattern occurrences. However, before proceeding further, let we describe briefly how string L is compressed and present some results relating the length of the compressed string Z with the entropy of the original text T.

BWT-based compressors. Let T^{bw} denote the last column L of the matrix $\mathcal{M}$, output of BWT. It is customary not to include the symbol # in T^{bw}, instead we store separately its position within the last column of $\mathcal{M}$. The reason why we should care about the string T^{bw} is that for any substring s of T, the characters preceding s in T (and thus having s as *following context*) are grouped together in a contiguous part of T^{bw}. This holds independently of the length of s, and indeed ensures that in any localized region of the transformed string we will likely find few distinct characters of Σ. The overall effect is that the probability that a given character c occurs at a given point in T^{bw} is very high if c occurs near to that point, and low otherwise. This property is exactly the one needed for effective compression by a move-to-front coder [7], which encodes an instance of a character c by the count of distinct characters seen since its last occurrence. On our running example, the move-to-front coder would produce the sequence: $\langle i, p, s, 0, m, 3, 0, 2, 0, 1, 0\rangle$ where characters are used to denote their first occurrences.

When we apply the move-to-front procedure to the string T^{bw} the resulting output, call it T^{mtf}, will be dominated by *low numbers*. As an example, if T is an English text T^{mtf} usually contains more than 50% zeroes. For this reason, the string T^{mtf} can be efficiently compressed with a Huffman or an arithmetic coder or more complex ad-hoc techniques [8, 38]. We point out that even the simplest BWT-based algorithms outperform widely used compressors such as `gzip` and `pkzip`. More advanced BWT-based compressors, such as `bzip2` [38], are among the best compressors currently available and achieve very good compression ratio using relatively small resources (time and space).

The promising experimental results have been recently followed by various theoretical studies both in the information theoretic setting (assuming T generated by a finite memory ergodic source) as well in the worst-case setting (assuming T as an arbitrary character sequence of a given length u). These results have shed some light on the positive features of BWT-based compressors and have shown that the BWT has remarkable theoretical properties not shared by other compression algorithms (see [26] and references therein).

In order to describe our results on searching in BWT-compressed files, we must commit ourselves to one of the several algorithms based on the BWT. The variant we consider, we call it `BW_RLX`, compresses the string T^{mtf} in two steps (recall that T^{mtf} is a string over the alphabet $0, 1, \ldots, |\Sigma| - 1$). In the first step the runs of zeroes produced by the MTF transformation are encoded as follows. Let $\mathbf{0}, \mathbf{1}$ denote two new symbols. For $m \geq 1$, let $Bin(m)$ denote the number $m + 1$ written in binary using $\mathbf{0}$ and $\mathbf{1}$, starting with the least significant bit first, and discarding the most significant bit. For example

$$Bin(1) = \mathbf{0}, \quad Bin(2) = \mathbf{1}, \quad Bin(3) = \mathbf{00}, \quad Bin(4) = \mathbf{10}, \quad \ldots.$$

This is known as `1/2` encoding because of the following property: if $Bin(m) = b_0 b_1 \cdots b_k$ then $m = \sum_{j=0}^{k} (b_j + 1) 2^j$. This encoding is *decodable on-line* because the decoding consists in translating each bit b_j to a sequence of $(b_j + 1) 2^j$ zeroes.

We define $T^{rl} = \mathtt{rle}(T^{mtf})$ as the string T^{rl} obtained from T^{mtf} by replacing each (maximal) run 0^m with $Bin(m)$. For example, if $T^{mtf} = 110022013000$, then $T^{rl} = 11\mathbf{1}22\mathbf{0}13\mathbf{00}$. Note that the string T^{rl} is defined over the alphabet $\{\mathbf{0}, \mathbf{1}, 1, 2, \ldots, |\Sigma| - 1\}$. The final step of the algorithm `BW_RLX` is the coding of T^{rl} using the prefix code `PC` defined as follows. The symbols $\mathbf{0}$ and $\mathbf{1}$ are coded using two bits (10 for $\mathbf{0}$, 11 for $\mathbf{1}$). The symbol i, $i = 1, 2, \ldots, |\Sigma| - 1$, is coded in $1 + 2 \lfloor \log(i + 1) \rfloor$ bits using a prefix code for the integer $i + 1$: $\lfloor \log(i + 1) \rfloor$ 0's followed by the binary representation of $i + 1$ which takes $1 + \lfloor \log(i + 1) \rfloor$ bits, the first one being a 1.

Summing up, `BW_RLX` compresses the string T by composing four procedures: `bwt` which produces $T^{bw} = \mathtt{bwt}(T)$, `mtf` which produces $T^{mtf} = \mathtt{mtf}(T^{bw})$, `rle` which produces $T^{rl} = \mathtt{rle}(T^{mtf})$, and `PC` which produces the final compressed string $Z = \mathtt{PC}(T^{rl})$. Despite of its simplicity `BW_RLX`'s compression ratio can be bounded in terms of the entropy of the input string. For any $k \geq 0$, the kth order *modified empirical entropy* H_k^* is the maximum amount of compression we can achieve using for each character a code which only depends on the k characters preceding it and with the additional requirement that the coding of a string T takes at least $\log |T|$ bits (see Appendix A or [25] for the formal definition). In [25, Sect. 5][2] it is shown that for any $k \geq 0$ there exists a constant g_k such that for any string T

$$|\mathtt{BW\_RLX}(T)| \leq 5 \ |T| \ H_k^*(T) + g_k. \tag{1}$$

This bound tells us that even for strings with very small entropy `BW_RLX`'s output size is within a constant factor of the optimum. At the same time, the algorithm `BW_RLX` is sufficiently simple so that in the rest of the paper we may concentrate on the working of the searching algorithm without being distracted by the details of the compression.

[2]The algorithm `BW_RLX` corresponds to the procedure `A`* described in [25]. Note that in order to achieve the bound (1) the BWT should be applied to the string T reversed.

Algorithm $\mathsf{BW\_Search}(P[1,p])$

1. $c = P[p]$, $i = p$;
2. $sp = C[c] + 1$, $ep = C[c+1]$;
3. **while** $((sp \leq ep)$ **and** $(i \geq 2))$ **do**
4. $\quad c = P[i-1]$;
5. $\quad sp = C[c] + \mathsf{Occ}(c, 1, sp-1) + 1$;
6. $\quad ep = C[c] + \mathsf{Occ}(c, 1, ep)$;
7. $\quad i = i - 1$;
8. **if** $(ep < sp)$ **then return** "not found" **else return** "found $(ep - sp + 1)$ occurrences".

Figure 2: Algorithm for computing the number of occurrences of $P[1,p]$ in $T[1,u]$.

4 Searching in BWT-compressed text

Let $T[1,u]$ denote an arbitrary text over the alphabet Σ, and let $Z = \mathtt{BW\_RLX}(T)$. In this section we describe an algorithm which, given Z and a pattern $P[1,p]$, reports all occurrences of P in the uncompressed text T without uncompressing the whole Z. Our algorithm makes use of the relation between the suffix array $\mathcal{A}$ and the matrix $\mathcal{M}$ (see Section 3). Recall that the suffix array $\mathcal{A}$ posses two very nice structural properties which are usually exploited to support fast pattern searches [23]: (i) all the suffixes of the text $T[1,u]$ prefixed by a pattern $P[1,p]$ occupy a contiguous portion (subarray) of $\mathcal{A}$; (ii) that subarray has starting position sp and ending position ep, where sp is actually the *lexicographic position* of the string P among the ordered sequence of text suffixes.

We start by describing an algorithm, called BW_Search, which identifies the positions sp and ep by accessing only the (transformed text) string T^{bw} and some auxiliary *array-based* data structures. This algorithm operates on the string produced by the BWT which is still uncompressed. Combining compression with indexing will be the next step of our solution, and will be detailed next.

4.1 First step: Searching in T^{bw}

Algorithm BW_Search consists of p phases each one preserving the following invariant: *At the i-th phase, the parameter sp points to the first row of $\mathcal{M}$ prefixed by $P[i,p]$ and the parameter ep points to the last row of $\mathcal{M}$ prefixed by $P[i,p]$.* The pseudo-code is given in Fig. 2. We start with $i = p$ (step 1), so that sp and ep can be determined via the array C (step 2). If $ep < sp$ then we can conclude that $P[p]$ does not occur in T and hence P does not too. After the final phase, sp and ep will delimit the portion of $\mathcal{M}$ (and thus of the suffix array $\mathcal{A}$) containing all text suffixes prefixed by P. The integer $(ep - sp + 1)$ will account for the number of occurrences of P in T.

Note that BW_Search makes use in Steps 5 and 6 of a subroutine $\mathsf{Occ}(c, 1, k)$ which reports the number of occurrences of the character c in $T^{bw}[1,k]$. The following lemma proves the correctness of BW_Search, assuming Occ works as claimed.

Lemma 2 *For $i = p, p-1, \ldots, 2$, let us assume that sp (resp. ep) stores the position of the first (resp. last) row in $\mathcal{M}$ prefixed by $P[i,p]$. If $P[i-1,p]$ occurs in T, then Step 5 (resp. Step 6) of* BW_Search *correctly updates the value of sp (resp. ep) thus pointing to the first (resp. last) row prefixed by $P[i-1,p]$.*

Proof: For the sake of presentation, let us refer to the matrix $\mathcal{M}$ of BWT and concentrate on the computation of sp. By the inductive hypothesis, sp points to the first row of $\mathcal{M}$ prefixed by the string $P[i,p]$. Let $\mathcal{M}[s] = cP[i,p]\alpha$ be the *first* row in $\mathcal{M}$ prefixed by $P[i-1,p]$ (if any), where α is an arbitrary string and $c = P[i-1]$. Indeed $\mathcal{M}[s]$ is the row we are interested in, so that we aim at proving that $s = C[c] + \mathsf{Occ}(c, 1, sp-1) + 1$.

We rotate cyclically to the left the row $\mathcal{M}[s]$ thus obtaining the row $\mathcal{M}[s'] = P[i,p]\alpha c$, for a proper value s'. By the hypothesis, sp is the first row of $\mathcal{M}$ prefixed by $P[i,p]$ and thus $sp \leq s'$. From Lemma 1, s' must be the first row of $\mathcal{M}$ which follows the position sp and ends with the character c (i.e. $L[s'] = c$). If this were not the case, then we would have found a row $\mathcal{M}[t']$ having the form $P[i,p]\beta c$ with $sp \leq t' < s'$. This way $\mathcal{M}[t']$ would precede lexicographically $\mathcal{M}[s']$ and hence β would be lexicographically smaller than α. The row $cP[i,p]\beta$ would exist in $\mathcal{M}$ and should occur before $\mathcal{M}[s]$, thus contradicting the definition of s.

From Lemma 1 we know that s is equal to $C[c]$ plus the rank of $\mathcal{M}[s']$ among all rows in $\mathcal{M}$ ending with the character c. This rank is actually given by $\mathsf{Occ}(c, 1, sp-1) + 1$, since it equals the number of times the character c occurs in the prefix $T^{bw}[1, sp-1]$, plus one unit. This proves the correctness of our formula for updating sp. A similar argument can be used to prove the correctness of the formula for ep, and easily show that the algorithm correctly detects the case in which $P[i-1,p]$ is not a prefix of any row in $\mathcal{M}$ (and therefore does not appear in T). ∎

The running time of the algorithm $\mathsf{BW\_Search}$ clearly depends on the cost of $\mathsf{Occ}(c,1,k)$ which, we next show, takes $O(1)$ time. The algorithm is slightly inspired to an idea of Munro [31] and runs on a RAM with word size $\Theta(\log u)$ bits.

We logically partition the transformed string T^{bw} into substrings of ℓ characters each (called *buckets*), and denote them by $BT_i = T^{bw}[(i-1)\ell+1, i\ell]$, for $i = 1, \ldots, u/\ell$. This partition naturally induces a partition of T^{mtf} into u/ℓ buckets $BT_1^{mtf}, \ldots, BT_{u/\ell}^{mtf}$ again of size ℓ. For the moment we assume that each run of zeroes in T^{mtf} is entirely contained in a single bucket and we describe our algorithm for computing $\mathsf{Occ}(c,1,k)$ under this simplifying assumption. Later we will discuss the general case in which a sequence of zeroes may span several buckets.

Under our assumption, the buckets BT_i^{mtf} induce a partition of the compressed file Z into u/ℓ *compressed buckets* $BZ_1, \ldots, BZ_{u/\ell}$, defined as $BZ_i = \mathtt{PC}(\mathtt{rle}(BT_i^{mtf}))$. By construction any compressed bucket BZ_i has size at most $\ell' = (1 + 2\lfloor \log \Sigma \rfloor)\ell$ bits.

Our algorithm for computing $\mathsf{Occ}(c,1,k)$ makes use of a hierarchical decomposition of $T^{bw}[1,k]$ in three substrings as follows: (i) the longest prefix of T^{bw} having length a multiple of ℓ^2 (i.e. $BT_1 \ldots BT_{i^*\ell}$, where $i^* = \lfloor \frac{k-1}{\ell^2} \rfloor$), (ii) the longest prefix of the remaining suffix having length a multiple of ℓ (i.e. $BT_{i^*\ell+1} \ldots BT_{i-1}$, where $i = \left\lceil \frac{k}{\ell} \right\rceil$), and finally (iii) the remaining suffix which is indeed a prefix of the bucket BT_i. This way $\mathsf{Occ}(c,1,k)$ can be computed as the sum of three quantities in $O(1)$ time, using a reduced space. More precisely, we keep the following auxiliary data structures:

- For the calculations on the string of point (i), we use:
  - For $i = 1, \ldots, u/\ell^2$, the array $NO_i[1, |\Sigma|]$ stores in each entry $NO_i[c]$ the number of occurrences of c in $BT_1, \ldots, BT_{i\ell}$.
  - The array $W[1, u/\ell^2]$ stores in each entry $W[i]$ the value $\sum_{j=1}^{i\ell} |BZ_j|$, that is, the sum of the sizes of the compressed buckets $BZ_1, \ldots, BZ_{i\ell}$.
- For the calculations on the string of point (ii), we use:

Algorithm $\mathsf{Occ}(c, 1, k)$

1. Let $i = \lceil k/\ell \rceil$, $j = k - (i-1)\ell$, $i^* = \lfloor (k-1)/\ell^2 \rfloor$.
2. Let $N_c = NO_{i^*}[c] + NO'_i[c]$.
3. Let $s_i = W[i^*] + W'[i]$ denote the position of the last bit of BZ_{i-1} within Z.
4. Let $n_c = S[c, j, Z[s_i + 1, s_i + \ell'], MTF[i]]$.
5. Return $N_c + n_c$.

Figure 3: Algorithm for computing the number of occurrences of c in $T^{bw}[1, k]$.

- For $i = 1, \ldots, u/\ell$, the array $NO'_i[1, |\Sigma|]$ stores in each entry $NO'_i[c]$ the number of occurrences of c in $BT_{i^*+1}, \ldots, BT_{i-1}$ (this string has length less than ℓ^2).
- The array $W'[1, u/\ell]$ stores in each entry $W'[i]$ the value $\sum_{j=i^*+1}^{i-1} |BZ_j|$, that is, the overall size of the compressed buckets $BZ_{i^*+1}, \ldots, BZ_{i-1}$ (bounded above by $O(\ell^2)$).

• For the calculations on the (compressed) buckets, we use:

- The array $MTF[1, u/\ell]$ stores in each entry $MTF[i]$ a picture of the state of the MTF-list at the beginning of the encoding of BT_i. Each entry takes $|\Sigma| \log |\Sigma|$ bits.
- The table S stores in each entry $S[c, h, BZ_i, MTF[i]]$ the number of occurrences of the character c among the first h characters of BT_i. The values of BZ_i and $MTF[i]$ are used to determine BT_i from Z. Table S has $O(\ell 2^{\ell'})$ entries each one requiring $\log \ell$ bits.

As a result, the calculation of $\mathsf{Occ}(c, 1, k)$ proceeds as indicated in the pseudo-code of Fig. 3 by exploiting the decomposition of $T^{bw}[1, k]$ in three substrings as described in points (i)–(iii) above. First, the bucket BT_i containing $c = T^{bw}[k]$ is determined via simple algebraic calculations, together with the position $j = k - (i-1)\ell$ of this character in BT_i and the parameter i^* (step 1). Then the number of occurrences of the character c in the prefix $BT_1, \ldots, BT_{i^*\ell}$ (point (i)) is determined via $NO_{i^*}[c]$, and the number of occurrences of c in the substring $BT_{i^*\ell}, \ldots, BT_{i-1}$ (point (ii)) is determined via $NO'_i[c]$ (step 2). Finally in steps 3-4, the compressed bucket BZ_i is retrieved from Z and the number of occurrences of c within $BT_i[1, j]$ are accounted using the table S. The sum of these three quantities gives $\mathsf{Occ}(c, 1, k)$.

Assume now that we choose $\ell = \Theta(\log u)$ so that $\ell' = c \log u$ with $c < 1$. Under this assumption every step of Occ consists of arithmetic operations or table lookup operations involving $O(\log u)$-bit operands. Consequently every call to Occ takes $O(1)$ time on a RAM with word size $\Theta(\log u)$ bits. As far as the space occupancy is concerned, the arrays NO and W take $O((u/\ell^2) \log u) = O(u/\log u)$ bits. The arrays NO' and W' take $O((u/\ell) \log \ell) = O((u/\log u) \log\log u)$ bits. The array MTF takes $O(u/\ell) = O(u/\log u)$ bits. Table S consists of $O(\ell 2^{\ell'})$ $\log \ell$-bit entries, and thus it occupies $O(2^{\ell'} \ell \log \ell) = O(u^c \log u \log\log u)$ bits, where $c < 1$. We conclude that the auxiliary data structures used by Occ occupy $O((u/\log u) \log\log u)$ bits (in addition to the compressed file Z).

We are now ready to discuss the general case in which T^{mtf} contains one or more runs of zeroes which cross the border of the buckets BT^{mtf}. Suppose for example that a sequence of $a+b$ zeroes is split between two buckets, let say BT_{i-1}^{mtf} ends with a zeroes and BT_i^{mtf} starts with b zeroes (a similar argument can be adopted in the case of a 0-sequence overlapping an entire bucket).

4.1 **if** c is the first character in $MTF[i]$ **then**

4.2 **if** $j \leq LT[i]$ **then** set $n_c = j$

4.3 **else** set $n_c = LT[i] + S[c, j - LT[i], Z[s_i + 1 + LZ[i], s_i + \ell'], MTF[i]]$

4.4 **else**

4.5 **if** $j \leq LT[i]$ **then** set $n_c = 0$

4.6 **else** set $n_c = S[c, j - LT[i], Z[s_i + 1 + LZ[i], s_i + \ell'], MTF[i]]$

Figure 4: Step 4 of procedure Occ revisited.

In order to define the corresponding compressed buckets BZ_{i-1} and BZ_i, we observe that `rle` transforms this sequence of zeroes into the string $\sigma = Bin(a + b)$. We interpret every digit of σ according to the `1/2` decoding scheme (see Section 2) and we assign to BZ_{i-1} the shortest prefix of σ whose `1/2` decoding is greater or equal than a. The remaining digits of σ are assigned to BZ_i. It should be clear that table look-up at Step 4 of Occ works correctly for BZ_{i-1} but does not work for BZ_i, because BZ_{i-1} is a faithful encoding of BT_{i-1}^{mtf} whereas the leading b zeroes of BT_i^{mtf} are usually not faithfully encoded by the digits of σ which we assigned to BZ_i. For this reason we introduce two additional arrays $LT[1, u/\ell]$ and $LZ[1, u/\ell]$ defined in such a way that the portion of BT_i^{mtf} starting from position $LT[i] + 1$ is faithfully encoded by the portion of BZ_i starting from position $LZ[i] + 1$ (in our example $LT[i] = b$). As a consequence, Step 4 of Occ now proceeds as shown in Figure 4.

Since the arrays LT and LZ occupy $O((u/\ell)\log \ell) = O((u/\log u)\log\log u)$ bits, their introduction does not change the asymptotic space occupancy of the algorithm. We can now state and prove the first important result of our paper.

Theorem 1 *Let Z denote the output of the algorithm* `BW_RLX` *on input $T[1, u]$. The number of occurrences of a pattern $P[1, p]$ in $T[1, u]$ can be computed in $O(p)$ time on a RAM. The space occupancy is $|Z| + O\left(\frac{u}{\log u}\log\log u\right)$ bits. The construction time of the data structures supporting the on-line search algorithm is $O(u)$, the construction space is $O(u \log u)$ bits.*

Proof: The algorithm BW_Search consists of a loop, which is executed at most $p - 1$ times, each iteration taking $O(1)$ time. The space occupancy of the data structures employed by algorithm BW_Search is dominated by that of Occ. The construction of the overall set of data structures takes $O(u)$ time and $O(u \log u)$ bits of space, provided that McCreight's algorithm [27] is employed to compute the matrix $\mathcal{M}$. ∎

4.2 Second step: Determining the occurrences

We now consider the problem of determining the exact position in the text $T[1, u]$ of all the occurrences of the pattern $P[1, p]$. This means that for $s = sp, sp + 1, \ldots, ep$, we need to find the position $pos(s)$ in T of the suffix which prefixes the sth row $\mathcal{M}[s]$. We propose two approaches, one is simple but slow, the other approach is faster and based on a sophisticated exploitation of a different compression algorithm for the string T^{bw}.

Our first, simple approach consists of *logically marking* the rows of $\mathcal{M}$ which correspond to text positions having the form $1 + i\eta$, for $i = 0, 1, \ldots, u/\eta$ and η proper parameter to be fixed

Algorithm $\mathsf{Retrieve}(sp, ep)$

1. Let $\eta = \Theta(\log^2 u)$. For $i = 0, 1, \ldots, u/\eta$, logically mark the rows of $\mathcal{M}$ which correspond to suffixes starting at positions $i\eta + 1$ in T.
2. Let $occ = ep - sp + 1$, where sp and ep are determined by algorithm $\mathsf{BW\_Search}(P[1,p])$.
3. **for** $j = 1, \ldots, occ$ **do**
4. $\quad x = \mathsf{Get\_pos}(sp + j - 1)$;
5. $\quad$ **print** "The jth pattern occurrence starts in T at position x";

Procedure $\mathsf{Get\_pos}(s)$

1. $s' = s$, $v = 0$;
2. **while** row $\mathcal{M}[s']$ is not *marked* **do**
3. $\quad c = T^{bw}[s']$;
4. $\quad m = \mathsf{Occ}(c, 1, s')$, $s' = C[c] + m$ (i.e. LF mapping);
5. $\quad v = v + 1$;
6. **return** $pos(s') + v$;

Figure 5: Algorithm for determining the exact positions where pattern $P[1,p]$ occurs in the uncompressed text $T[1,u]$. Step 1 is executed at compression time.

later. We store with these marked rows the starting positions of the corresponding text suffixes explicitly. This preprocessing is done at compression time. We then find $pos(s)$ as follows (see the pseudo-code in Fig.5). If s is a marked row, then there is nothing to be done and its position is directly available. Otherwise, we use the LF mapping to find the row s' corresponding to the suffix $T[pos(s)-1, u]$ (Step 4). We iterate this procedure v times until s' points to a marked row; then we sum up to the position $pos(s')$, which is directly available, the number v so that $pos(s) = pos(s')+v$. By construction this requires at most η iterations, each taking constant time (Steps 2–5). The only non trivial operation of each iteration is step 3 which can be implemented with at most 2Σ calls to procedure Occ: $c = T^{bw}(s)$ is the only character for which $\mathsf{Occ}(c, 1, s) \neq \mathsf{Occ}(c, 1, s-1)$.

We now show that letting $\eta = \Theta(\log^2 u)$, algorithm $\mathsf{Retrieve}$ takes $O(u/\log u)$ bits of additional space. The main issue is to implement Step 1, that is, the logical marking of the rows of $\mathcal{M}$ corresponding to the text suffixes starting at positions $1 + i\eta$, $i = 0, \ldots, u/\eta$. Our solution consists of storing the row numbers in a two-level bucketing scheme, like the one used in the previous section. This occupies $O((u/\eta)\log u)$ bits and makes it possible to check in $O(1)$ time whether a certain row is logically marked. For each marked row we also keep the starting position of the corresponding suffix in T, which requires additional $O(\log u)$ bits per marked row. Consequently, the overall space required for that logical marking is $O((u/\eta)\log u) = O(u/\log u)$ bits.

Theorem 2 *Algorithm* $\mathsf{Retrieve}$ *lists all the occ pattern occurrences in* $O(occ \log^2 u)$ *time and additional space* $O(\frac{u}{\log u})$ *bits.*

We are now ready to state our first complete result, obtained by combining Algorithm $\mathsf{BW\_Search}$, Algorithm $\mathsf{Retrieve}$ and equation (1).

Theorem 3 *Given a text $T[1,u]$, we can preprocess it in $O(u)$ time so that all the occ occurrences of a pattern $P[1,p]$ in T can be listed in $O(p + occ \log^2 u)$ time on a RAM. The space occupancy is bounded by $5H_k^*(T) + O(\frac{\log\log u}{\log u})$ bits per input symbol, for any fixed $k \geq 0$.*

The slowness of the previous algorithm is due to the backward movement in T of one character at a time, which therefore requires $\Theta(\log^2 u)$ steps overall in the worst case. We now show how to introduce some *shortcuts* which allow to move faster on T jumping by more than one character at a time. As a result, we get an algorithm which still requires $o(u)$ bits of storage on a compressible text T and achieves $O(\log^\epsilon u)$ time cost per occurrence retrieval, for any fixed $\epsilon > 0$.

The primary ingredient of our new approach is a procedure for computing the LF-mapping over a string $\bar{T}[1,v]$ drawn from an alphabet Λ of *non-constant* size. Let $\bar{T}^{bw} = \mathtt{bwt}(\bar{T})$. For each $j = 0, 1, \ldots, 2\log\log v$, we construct the string $\bar{T}_j^{bw}$ by selecting from $\bar{T}^{bw}$ one out of 2^j consecutive occurrences of α, for every character $\alpha \in \Lambda$. Moreover, we logically mark the characters of $\bar{T}_j^{bw}$ which occur in the next string $\bar{T}_{j+1}^{bw}$. Notice that $\bar{T}_0^{bw} = \bar{T}^{bw}$ and that the length of $\bar{T}_j^{bw}$ is $\Theta(\frac{v}{2^j})$.

Now consider a modified move-to-front procedure, called $\mathtt{mtf}^*$, that encodes a character in $\bar{T}^{bw}$ by the *number of characters seen* since its previous occurrence[3]. For $j = 0, 1, \ldots, 2\log\log v$, let $\bar{T}_j^{mtf^*} = \mathtt{mtf}^*(\bar{T}_j^{bw})$. We store the strings $\bar{T}_0^{mtf^*}, \bar{T}_1^{mtf^*}, \ldots, \bar{T}_{2\log\log v}^{mtf^*}$ and, for the characters in $\bar{T}_{2\log\log v}^{bw}$, we also keep the explicit value of their LF-mapping in the original transformed text $\bar{T}^{bw}$.

The high-level idea to compute $LF[s]$ for a character $\alpha = \bar{T}^{bw}[s]$ is to use the arrays $\bar{T}_j^{mtf^*}$'s to jump, in $\Theta(\log\log v)$ steps, to an occurrence of α which is also in $\bar{T}_{2\log\log v}^{bw}$. For this occurrence we have the explicit LF-mapping and we can easily reconstruct $LF[s]$ via Lemma 3. More precisely, we start from $\bar{T}_0^{bw}[s]$ and we move to $\bar{T}_1^{bw}, \bar{T}_2^{bw}, \bar{T}_{2\log\log v}^{bw}$ updating a counter v as follows. Let $\bar{T}_j^{bw}[i]$ be the currently examined character (initially $j = 0$, $i = s$ and $v = 0$); if this character α is marked then we are done and v remains unchanged; otherwise we move to the previous copy of α in $\bar{T}_j^{bw}$ using the distance encoded in $\bar{T}_j^{mtf^*}[i]$ and sum 2^j to the counter v. Note that we are indeed implicitly jumping 2^j occurrences of the character α in the original $\bar{T}^{bw}$, and thus we account for them in v. We have now reached a marked character α, thus we can move to its corresponding copy in $\bar{T}_{j+1}^{bw}$. After $\Theta(\log\log v)$ iterations, we reach a copy of α in $\bar{T}_{2\log\log v}^{bw}$ for which its LF-mapping over $\bar{T}^{bw}$ is available. Consequently, we sum this value with v and thus obtain $LF[s]$ in accordance to Lemma 1.

Note that the in above scheme we do not need the arrays $\bar{T}_j^{bw}$'s: we only need to access the values stored in $\bar{T}_0^{mtf^*}, \bar{T}_1^{mtf^*}, \ldots, \bar{T}_{2\log\log v}^{mtf^*}$. This is crucial to achieve a sublinear space bound. Let $\mathtt{bin}(i)$ denote the binary representation of $i+1$ (which takes $1 + \lfloor \log(i+1) \rfloor$ bits). We construct the following three arrays for each $j = 0, 1, \ldots, 2\log\log v$:

- Z_j is a string succinctly encoding $\bar{T}_j^{mtf^*}$ and consisting of the concatenation $\mathtt{bin}(\bar{T}_j^{mtf^*}[1])\mathtt{bin}(\bar{T}_j^{mtf^*}[2]) \cdots \mathtt{bin}(\bar{T}_j^{mtf^*}[v/2^j])$;
- W_j is a binary array of length $|Z_j|$ such that $W_j[i] = 1$ iff $Z_j[i]$ is the starting bit-position of the encoding of a value in $\bar{T}_j^{mtf^*}$;
- N_j is a binary array of length $|Z_j|$ such that $N_j[i] = 1$ iff $W_j[i] = 1$ and the corresponding character of $\bar{T}_j^{bw}$ occurs also in $\bar{T}_{j+1}^{bw}$ (and thus it is marked).

Using the procedures `rank` and `select` described in [31] and applied on the arrays above, we can implement each step of the computation of $LF[s]$ described before in $O(1)$ time. Namely,

[3]Notice that $\mathtt{mtf}^*$ is different from $\mathtt{mtf}$ because in the latter we count the number of *distinct* characters.

given an encoding of $\bar{T}_j^{mtf^*}$ starting at bit position i, we can check if this corresponds to a *marked* character α by accessing $N_j[i]$. If this is the case we count the number of characters of $\bar{T}_{j+1}^{bw}$ preceding α (via `rank` over $N_j[1,i]$) and finally find the position of the encoding of α in $\bar{T}_{j+1}^{mtf^*}$ (via `select` on N_{j+1}). If α is not marked, then we decode in $O(1)$ time the entry of $\bar{T}_j^{mtf^*}$ starting at the ith bit of Z_j and then jump to the previous occurrence of α in $\bar{T}_j^{bw}$ via `rank` and `select` operations over array W_j. In summary, each of these operations takes $O(1)$ time and requires a set of auxiliary data structures which occupies $O(|Z_0| + |Z_1| + \cdots + |Z_{2\log\log v}|)$ bits of storage. The space occupancy induced by the explicit LF-mapping kept for the characters in $\bar{T}_{2\log\log v}^{bw}$ is $O(v/\log v)$ bits (we store $O(v/\log^2 v)$ entries of $O(\log v)$ bits each).

Lemma 3 *Given a string $\bar{T}[1,v]$ over an arbitrary alphabet Λ, we can compute the LF-mapping over $\bar{T}^{bw}$ in $O(\log\log v)$ time and $O\Big(v(1+H_k(\bar{T})) + |\Lambda|\log v \log\log v\Big)$ bits of storage*[4].

Proof: The time bound follows by the discussion above. To prove the space bound we first observe that, by [7], we know that $|Z_j| \le (v/2^j)(1 + H_0(\bar{T}_j^{mtf^*})) + |\Lambda|\log v$. In addition, it is easy to see that $H_0(\bar{T}_{j+1}^{mtf^*}) \le H_0(\bar{T}_j^{mtf^*}) + (v/2^{j+1})|\Lambda|\log v$. Hence, the space occupancy is bounded by $O(v(1 + H_0(\bar{T}^{mtf^*})) + |\Lambda|\log v\log\log v)$ bits. The thesis follows by Theorem 4.1 in [25]. ■

We now go back to our string $T[1,u]$ over the constant size alphabet Σ, and we show how to determine the position $pos(s)$ of the text suffix prefixing the row $\mathcal{M}[s]$ in sublogarithmic time. First, we show how to achieve the time bound $O(\log^{(1/2)+\epsilon} u)$, then refine this result achieving the final $O(\log^\epsilon u)$ time bound.

The first algorithm works as follows. At compression time we logically mark the rows of $\mathcal{M}$ which correspond to text positions of the form $1 + i\gamma$ for $i = 0, \ldots, u/\gamma$ and $\gamma = \Theta(\log^{(1/2)+\epsilon} u)$. Then, we consider the string T_0 obtained by grouping the characters of T in blocks of size γ. Clearly, T_0 has length u/γ and its characters belong to the alphabet Σ^γ. Let $\mathcal{M}_0$ denote the cyclic-shift matrix associated to T_0, hence $\mathcal{M}_0$ consists of the marked rows of $\mathcal{M}$. We preprocess T_0 according to Lemma 3, and mark the rows which correspond to the suffixes of T_0 starting at the positions $1 + i\eta$, for $i = 0, \ldots, |T_0|/\eta$ and $\eta = \Theta(\log^{(1/2)+\epsilon} u)$. For these rows we keep the starting position of the corresponding text suffix. To compute $pos(s)$ we first iterate the **while** loop of procedure Get_pos until we reach a marked row s'. The value $pos(s')$ is computed by finding its corresponding row in $\mathcal{M}_0$, and then using the procedure Get_pos with the only modification that at Step 4 the LF-mapping is computed via the procedure analyzed in Lemma 3. Summing up, the computation of $pos(s)$ takes $O(\gamma)$ time (for finding a marked row in T) plus $O(\eta\log\log u)$ time (for finding a marked row in T_0 for which we have the starting position of the corresponding text suffix explicitly). As far as the space occupancy is concerned, we observe that the marking of T and the counting of the number of marked rows in $\mathcal{M}$ that precede a given marked row s' (this is required in order to determine the position in $\mathcal{M}_0$ of $\mathcal{M}[s']$) can be done in constant time and $O(\frac{u}{\gamma}\log\log u)$ bits of storage. [5] Moreover, $\Theta(|T_0|/\eta)$ rows of $\mathcal{M}_0$ keep explicitly the

[4] Here we are using the empirical entropy H_k rather than the *modified* empirical entropy H_k^*. See Appendix A for the definitions.

[5] The idea is to define a two-level data structure in which we partition the rows of $\mathcal{M}$ into buckets of size $\Theta(\log^2 u)$ each. For each bucket, we take all the marked rows lying into it, and store them into a Packet B-tree [3] using as a key their distance from the beginning of the bucket. Since a bucket contains at most $O(\log^2 u)$ keys, each long $O(\log\log u)$ bits, membership queries take $O(1)$ time. The overall space is $O(\frac{u}{\gamma}\log\log u)$ bits. For each bucket we also store the number of marked rows in $\mathcal{M}$ preceding it, and for each marked row of $\mathcal{M}$ we also keep the number of marked rows preceding it into its bucket. This information occupies $O(\frac{u}{\gamma}\log\log u)$ overall bits. Using this two-level scheme, the counting of the number of marked rows in $\mathcal{M}$ preceding a given marked row s' can be done in $O(1)$ time.

starting position of the corresponding text suffix (see above), so that their space occupancy is $\Theta(|T_0| \log u/\eta) = \Theta(u/\log^{2\epsilon} u)$ bits. Finally, the space occupancy of the procedure for computing the LF-mapping in T_0^{bw} is given by Lemma 3. Summing up, the overall space occupancy is

$$O\left(\frac{u}{\gamma}(1 + H_k(T_0)) + |\Sigma|^{\gamma} \log u \log\log u + \frac{u \log u}{\gamma\eta}\right).$$

Since $H_k(T_0) \leq \gamma\ H_{k\gamma}(T) \leq \gamma\ H^*_{k\gamma}(T)$ a simple algebraic calculation yields that the overall space occupancy is $O\Big(H_k^*(T) + \frac{1}{\log^{2\epsilon} u}\Big)$ bits per input symbol, for any fixed k. The time bound is $O(\gamma + \eta \log\log u) = O(\log^{(1/2)+\epsilon} u \log\log u)$.

To compute $pos(s)$ in $O(\log^{\epsilon} u)$ time we take $\gamma = \Theta(\log^{\epsilon} u)$, and we apply the procedure for computing the LF-mapping in T_0 for $\Theta(\log^{\epsilon} u)$ steps, thus identifying a row of the matrix $\mathcal{M}_0$ prefixed by a text suffix starting at a position multiple of $L_1 = \Theta(\log^{2\epsilon} u)$. We cannot store the value of the starting position of the text suffix prefixing this row explicitly because it would require too much space, but we use the rows of this form (i.e. prefixed by suffixes starting at text positions $1 + iL_1$) to define a new text T_1 and a corresponding matrix $\mathcal{M}_1$ on which we iterate the approach above for $1/\epsilon = \Theta(1)$ times, each taking $\Theta(\log^{\epsilon} u \log\log u)$ time complexity. The space occupancy is dominated by the one required for T_0, $\mathcal{M}_0$ and their marking.

Theorem 4 *A text $T[1, u]$ can be indexed so that all the occ occurrences of a pattern $P[1, p]$ in T can be listed in $O(p + occ \log^{\epsilon} u)$ time on a RAM. The space occupancy is $O(H_k^*(T) + \frac{\log\log u}{\log^{\epsilon} u})$ bits per input symbol, for any fixed $k \geq 0$.*

5 Dynamizing our approach

Let Δ be a dynamic collection of texts $\{T_1, T_2, \ldots, T_m\}$ having arbitrary lengths and total size u. Collection Δ may shrink or grow over the time due to insert and delete operations which allow to add or remove from Δ an individual text string. Our aim is to store Δ in succinct space, perform the update operations efficiently, and be able to support the query: search for the occurrences of an arbitrary pattern $P[1, p]$ into Δ's texts. This problem can be solved in optimal time complexity and $O(u \log u)$ bits of storage using the suffix tree [27]. In the external memory setting, this problem can be solved [13] with effective I/O-bounds occupying $O((u \log u)/B)$ disk pages, where B is the disk page size expressed in bits. In the present section we aim at *dynamizing* our compressed index in order to keep Δ in a reduced space, which is still converging asymptotically to the entropy of the text collection, and be able to efficiently support update and search operations. Our result extends an elegant technique proposed at the beginning of '80 in [35, 28], here adapted to manage items (i.e. texts) of variable lengths.

We point out that in the literature some authors (see e.g. [30]) have investigated the design of *implicit* data structures for the dynamic dictionary problem, which is indeed a simpler version of the previous one. They achieved *sublinear auxiliary space* occupancy (i.e. $u \log_2 u + o(u)$ bits) and polylogarithmic time complexity for the various operations supported. In this paper we aim at surpassing the $\Omega(u)$ space bound and achieve a stronger information-content space bound, still supporting effective operations. In the following we bound the space occupancy of our data structure in terms of the entropy of the concatenation of Δ's texts. A better overall space reduction might possibly be achieved by compressing separately the texts T_i's. However, if the texts T_i's have similar statistics the entropy of the concatenated string is a reasonable lower bound to the compressibility of the collection. Furthermore, in the probabilistic setting where we assume that every text is generated by the same probabilistic source, the entropy of the concatenated string coincides with the entropy of the single texts and therefore provides a tight lower bound to the compressibility of the collection.

We first discuss the case in which Δ may change under only insert operations, and then turn our attention to the more general fully-dynamic case where inserts and deletes are allowed. We can focus on the situation in which the length p of the searched pattern is $O(\frac{u}{\log^2 u})$ because for the other range of p's values, the search operation can be implemented in a brute-force way by first decompressing the text collection and then searching for P into it using a scan-based string matching algorithm thus taking $O(p \log^2 u + occ)$ time complexity.

We partition the texts T_i's into $\eta = \Theta(\log^2 u)$ collections $\mathcal{C}_1, \ldots, \mathcal{C}_\eta$, each containing texts of overall length $O(\frac{u}{\log^2 u})$. This is always possible, independently of the lengths of the text strings in Δ, since the upper bound on the length of the searchable patterns allows us to split very long texts (i.e. texts of lengths $\Omega(\frac{u}{\log^2 u})$) into $2\log^2 u$ pieces overlapping for $\Theta(\frac{u}{\log^2 u})$ characters. This covering of a single long text with many shorter ones still allows us to find the occurrences of the searched patterns.

Every collection $\mathcal{C}_h$ is then partitioned into a series of subsets $\mathcal{S}_i^h$ defined as follows: $\mathcal{S}_i^h$ contains some texts of $\mathcal{C}_h$ having overall length in the range $[2^i, 2^{i+1})$, where $i = O(\log u)$. Each set $\mathcal{S}_i^h$ is simultaneously indexed and compressed using our opportunistic data structure (Theorem 4).

Searching for an arbitrary pattern $P[1,p]$, with $p = O(\frac{u}{\log^2 u})$, into the text collection Δ can be performed by searching for P in all the subsets $\mathcal{S}_i^h$ via the compressed index built on each of them, thus requiring overall $O(p \log^3 u + occ \log^\epsilon u)$ time in the worst case.

Inserting a new text $T[1,t]$ into Δ consists of inserting T into one of the sets $\mathcal{C}_h$, the most empty one. Then, a subset $\mathcal{S}_i^h$ is selected, where $i = \lfloor \log t \rfloor$, and T is inserted into it using the following approach. If $\mathcal{S}_i^h$ is empty then the compressed index is built for T and associated to this subset, thus taking $O(t)$ time. Otherwise the new set $\mathcal{S}_i^h \cup \{T\}$ is formed and inserted in $\mathcal{S}_{i+1}^h$. If the latter subset is full then the insertion process is propagated until an empty subset $\mathcal{S}_{i+j}^h$ is found. At this point, the compressed index is built over the set $\mathcal{S}_i^h \cup \ldots \cup \mathcal{S}_{i+j-1}^h \cup \{T\}$, by concatenating all the texts contained in this set to form a unique string, texts are separated by a special symbol (as usual). By noticing that these texts have overall length $\Theta(2^{i+j})$, we conclude that this propagation process has a complexity proportional to the overall length of the *moved* texts. Although each single insertion may be very costly, we can amortize this cost by charging $O(\log u)$ credits per text character (since $i, j = O(\log u)$), thus obtaining an overall amortized cost of $O(t \log u)$ to insert $T[1,t]$ in Δ. In fact each time T moves from one subset $\mathcal{S}_i^h$ to the next one $\mathcal{S}_{i+1}^h$, $\Theta(t)$ of its associated credits are used to pay for the reconstruction of the compressed index; all texts in $\mathcal{S}_i^h$ indeed contribute to pay for the overall reconstruction process so that we have enough credits to perform that task.

Some care must be taken to evaluate the space occupied during the reconstruction of the set $\mathcal{S}_i^h$. Here the collections $\mathcal{C}_h$ come into play. In fact, the construction of our compressed index over the set $\mathcal{S}_i^h$ requires the use of the suffix tree data structure (to compute the BWT) and thus $O(2^i \log 2^i)$ bits of auxiliary storage. This could be too much, but we ensured that every collection $\mathcal{C}_h$ contains texts having overall length $O(\frac{u}{\log^2 u})$. So that at most $O(\frac{u}{\log u})$ bits suffices to support any reconstruction process.

Theorem 5 *Let Δ be a dynamic collection of texts $\{T_1, T_2, \ldots, T_m\}$ having total length u. All the occ occurrences of a pattern $P[1,p]$ in the texts of Δ can be listed in $O(p \log^3 u + occ \log^\epsilon u)$ time in the worst case. Operation* insert *adds a new text $T[1,t]$ in Δ taking $O(t \log u)$ amortized time. The space occupancy is $O(H_k^*(\Delta)) + o(1)$ bits per input symbol, for any fixed $k \geq 0$.*

In the fully dynamic case also deletions must be supported on the collection Δ. The main problem here is that, from one side we would like to physically cancel the deleted texts, in order to avoid the listing of *ghost* occurrences belonging to texts no longer in Δ; but from the other side a physical deletion would be too much time-consuming to be performed on-the-fly. Amortization can still be used but much care must be taken when answering a query to properly deal with texts which have been *logically* deleted from $\mathcal{S}_i^h$'s.

For the sake of presentation let T^{bw} be the BWT of the texts stored in some $\mathcal{S}_i^h$. Some of these texts have been *logically deleted* by previous delete operations. To keep track of them, we denote by $\mathcal{I}_i^h$ the set of intervals of positions in T^{bw} occupied by deleted text suffixes. The underlying idea is that if a pattern occurrence is found in T^{bw} using our compressed index, we can immediately check if it is a real or a ghost occurrence by testing if that position occurs in any interval of $\mathcal{I}_i^h$. For simplicity of exposition, we store $\mathcal{I}_i^h$ into a simple balanced search tree thus introducing a $O(\log u)$ time slowdown in the cost of the listing of the pattern occurrences (other more efficient solutions could be employed but this would not change substantially the final complexity).

Every time a text $T[1,t]$ must be deleted from a subset $\mathcal{S}_i^h$, we search for all of its suffixes in $\mathcal{S}_i^h$, using the corresponding compressed index in $O(t)$ time, and then update accordingly $\mathcal{I}_i^h$ in $O(t \log u)$ time. The additional space required to store the balanced search tree is $O(|\mathcal{I}_i^h| \log u) = O(\frac{u}{\log u})$ bits, where we are assuming to physically delete the texts from $\mathcal{S}_i^h$ whenever a fraction of $\Theta(\frac{1}{\log^2 u})$ suffixes has been logically marked. Hence, each set $\mathcal{S}_i^h$ may undergo $O(\log^2 u)$ reconstructions before it shrinks enough to move back to the previous set $\mathcal{S}_{i-1}^h$. This implies that the amortized cost of delete is $O(t \log u + t \log^2 u) = O(t \log^2 u)$, where the first additive term denotes the cost of $\mathcal{I}_i^h$'s update and the second term accounts for the credits to be left in order to pay for the physical deletions.

One important point has been left out. How a text to be deleted is identified ? We may imagine to append to every text in Δ an identifier of $O(\log u)$ bits, and to keep track of the subset $\mathcal{S}_i^h$ containing a text via a table. The former information introduces an overhead of $O(m \log u)$ bits in the text representation, whereas the latter adds $O(m \log \log u)$ bits. This overhead is $o(u)$ if we reasonably assume that the texts are not too short, i.e. $\omega(\log u)$ bits each.

Theorem 6 *Let Δ be a dynamic collection of texts $\{T_1, T_2, \ldots, T_m\}$ having total length u. All the occ occurrences of a pattern $P[1,p]$ in the texts of Δ can be listed in $O(p \log^3 u + occ \log u)$ time in the worst case. Operation* insert *adds a new text $T[1,t]$ in Δ taking $O(t \log u)$ amortized time. Operation* delete *removes a text $T[1,t]$ from Δ taking $O(t \log^2 u)$ amortized time. The space occupancy is $O\Big(H_k^*(\Delta) + m\frac{\log u}{u}\Big) + o(1)$ bits per input symbol, for any fixed $k \geq 0$.*

6 A simple application: adding compression to Glimpse

Glimpse [24] is an indexing tool for file systems which achieves very small index sizes — in most cases $2-4\%$ of the original text size — and still allows very flexible full-text retrieval including boolean queries, approximate matching and regular expressions. These properties made it commonly used in more than a hundred of institutions and companies (see `http://webglimpse.org/otherusers.html`).

Glimpse is a hybrid between inverted files and scan-based approaches with *no* index. It relies on the observation that there is no need to index every word with an exact location (as it occurs in inverted files); but only pointers to an area where the word occurs (called a *block*) should be maintained. Glimpse assumes that the text $T[1,u]$ is *logically* partitioned into r blocks of size b each, and thus its index consists of two parts: a *vocabulary* V containing all the distinct words of the text; and for each word $w \in V$, a *list $L(w)$ of blocks* where the word w occurs. This blocking scheme induces two space savings: each word occurrence is represented in $\log_2(u/b)$ bits, instead of the standard $\log_2 u$ bits; and many occurrences of the same word w in a single block are represented only once in $L(w)$. Typically, 1 byte is enough for block pointers and $2-4\%$ is the index occupancy for medium sized text collections [24].

Given this index structure, the search scheme proceeds in two steps: first the queried word w is searched into the vocabulary, then all candidate blocks in $L(w)$ are *sequentially* examined to find all the w's occurrences. More complex queries (e.g. approximate or regular expression searches) can be supported by using *Agrep* [40] both in the vocabulary and in the block searches. Clearly, the search is effective if the vocabulary is small (which is typically the case [24]), if the query is enough selective (few text blocks are sequentially accessed), and if the block size is not too large (so that sequential block search is fast). The first two requirements are usually met in practice, so that the main constraint to the effective use of Glimpse remains the strict relation between block-pointer sizes and text sizes. Theoretical and experimental analysis of such block-addressing schemes [24, 5, 4] have shown that the Glimpse approach is effective only for medium sized texts (roughly up to 200Mb).

In order to make the blocks longer and thus overcome the limitation on the text size, recent papers tried to speed up their sequential scan by compressing each text block individually, and then searching it via proper opportunistic string-matching algorithms [24, 34]. The experimental results showed an improvement of about $30-50\%$ in the final performance, and a comparable reduction in the overall space occupancy. This implicitly shows that the second searching step dominates Glimpse's query performance. Our aim here is to overcome that bottleneck without compromising the good features of the Glimpse's index: flexibility, simplicity and succinct space occupancy of the internal-memory index. Surprisingly enough, the compressed index introduced in this paper naturally fits in the block-addressing framework and allows us to extend its applicability to larger text databases. The idea consists of two simple modifications to the Glimpse settings:

- Each block of the text is indexed and compressed via our opportunistic data structure.
- The search for a word w proceeds in two steps. First, the vocabulary V is sequentially scanned with the *Agrep* tool to find the list $L(w)$ of w's block occurrences. Subsequently, each block of $L(w)$ is accessed by means of its compressed index, and all the w's occurrences are listed.

This approach is named *Compressed Glimpse*, shortly `CGlimpse`, and has three main advantages:

- Each candidate block does not need to be scanned sequentially, and thus there is no longer a *linear dependence* between the block size and the search complexity. As a result, when dealing with large text collections, we can still use few blocks of larger sizes (hence small block pointers), without much affecting the overall search performance.
- A complex query $\mathcal{Q}$ (e.g. approximate query or regular expression search) can be performed efficiently by means of *exact pattern searches* on our compressed index. The idea is to use *Agrep* to solve $\mathcal{Q}$ on the vocabulary V, and thus turn a complex query into a collection of exact-match queries. All words w_i resolving $\mathcal{Q}$ in V are searched *exactly* in the blocks $L(w_i)$ via the corresponding compressed index.
- The construction of a unique suffix array over a huge text would be too much time consuming. With our new approach, we can build the compressed index on every block efficiently, preserve the good compression ratios of BWT-based algorithms (typically these algorithms operate on blocks of few megabytes [38]), and still search huge text collections effectively.

In some sense, `CGlimpse` is a compromise between a full-text index (like a suffix array) and a word-based index (like an inverted list) over a compressed text. A theoretical investigation of the performance of `CGlimpse` is yet feasible using a model generally accepted in Information Retrieval [5]. The first assumption is the Heaps law [16] which states that in a text of size u the number of distinct words grows as $V = O(u^{\beta})$ with $0 < \beta < 1$. Typical experimental values of β [15] are in the range $[0.4, 0.6]$, so that the vocabulary has size about $\sqrt{u}$ (in practice this corresponds to few megabytes). The second assumption is the Zipf's law [41] which states that if the words of the vocabulary are sorted in decreasing order of frequency, then the frequency of the ith word is $u/(i^{\theta} H_V^{(\theta)})$, where $H_V^{(\theta)} = \sum_{j=1}^{|V|} 1/i^{\theta}$ is a normalization term and θ is a parameter larger than 1. Typical experimental values of θ [15] are in the range $[1.7, 2.0]$. The last assumption is that in a text of size u, the number of matches for a given word with $k \geq 1$ errors grows as $\Theta(u^{\rho})$, with $\rho < 1$ (typical values are $\rho = 0.7$ [5]).

Theorem 7 *Assume that the text $T[1,u]$ is split into r blocks of size $b = \omega(1)$ each. In the above model both the size of* `CGlimpse`*'s index and the average time of a complex query are sublinear in u.*

Proof: The Zipf's law combined with the pessimistic assumption that the occurrences of a word are distributed uniformly along the text, allows us to prove that the index size is $O(rb^{1/\theta})$. Exactly the same bound as for Glimpse [5].

Let $occ(w)$ be the number of occurrences of a complex query on a word w. Retrieving all those occurrences takes $O(u^{\beta})$ time to look into the vocabulary plus $O(occ \log b)$ time to retrieve the positions in the compressed blocks (reasonably assume $|w| = O(1)$). Consequently, the averaged searching time is sublinear due to the assumption that $occ = O(u^{\rho})$ with $\rho < 1$. ■

The notable feature of the block-addressing scheme above is that it can achieve both sublinear space overhead and sublinear query time, whereas inverted indexes pointing to words or documents achieve only the second goal [4]. Unfortunately, Glimpse and its variants achieve this goal only under some restrictive conditions on the model parameters. To be precise [5], the query time of Glimpse is $O(u^{\beta} + occ\ b) = O(u^{\beta} + u^{\rho}\ b)$ under the model assumptions and its space occupancy is $O(rb^{1/\theta}) = O(\frac{u}{b^{1-1/\theta}})$. Hence if the classical Glimpse has to achieve sublinear space and time, then proper conditions on the block size b must be imposed. Conversely `CGlimpse` achieves time and space sublinearity without imposing any conditions on the model parameters.

As an illustrative example, let us consider the WSJ collection (Wall Street Journal) in the TREC archive [15] summing up to $u = 250$Mb. Here $\beta = 0.4, \theta = 1.87$ and $\rho = 0.8$ (for approximate queries up to 3 errors) [5]. Substituting to the formulas above and choosing $r = 256$ (like in the classical Glimpse index), we derive that both Glimpse and `CGlimpse` occupy $O(u^{0.53})$ space on the average, but their query time is on the average $\Theta(u)$ and $O(u^{0.8} \log u)$ respectively. Indeed `CGlimpse` achieves a *polynomial* speed up.

7 Conclusions

Various issues remain still to be investigated in various models of computation. In the RAM it would be interesting to avoid the $o(\log u)$ overhead incurred in the listing of the pattern occurrences. In an external-memory setting it would be interesting to devise a compressed index which takes advantage of the blocked access to the disk and thus reduces the I/O-complexity of the search operation by a factor $\Theta(B)$.

Another interesting issue concerns with the design of word-based indexes where the searches are mainly confined to words or prefixes of words. We are investigating a novel word-based compression scheme which takes advantage of this special setting and thus improves both the compression ratio and the search complexity. It allows to compress the pattern before searching it into the opportunistic data structure thus achieving $o(p)$ I/O-complexity.

We plan to investigate experimentally how our compressed index compares against inverted lists, suffix arrays and suffix trees. Our approach offers various practical advantages: it is array based, only a small portion of the text is decompressed at searching time, the compression algorithm is BWT-based and thus it should result superior to Huffman- or LZ-based schemes [37, 39]. We expect that from one hand, our compressed index will occupy space similar to the one required by inverted lists, but with the advantage of supporting full-text searches; and from the other hand, it will offer the same query operations of suffix trees and suffix arrays but with advantage of occupying much less space.

A Appendix: the empirical entropy of a string

In this appendix we review the definition of *empirical entropy* and *modified empirical entropy* of a string. Further details are given in [25]. Let s be a string of length n over the alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_h\}$, and let n_i denote the number of occurrences of the symbol α_i inside s. The zeroth order empirical entropy of the string s is defined as

$$H_0(s) = -\sum_{i=1}^{h} \frac{n_i}{n} \log\left(\frac{n_i}{n}\right), \tag{2}$$

where we assume $0 \log 0 = 0$. The value $|s|H_0(s)$, represents the output size of an ideal compressor which uses $-\log \frac{n_i}{n}$ bits for coding the symbol α_i. It is well known that this is the maximum compression we can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. We can achieve a greater compression if the codeword we use for each symbol depends on the k symbols preceding it. For any length-k word $w \in \Sigma^k$ let w_s denote the string consisting of the characters following w inside s. Note that the length of w_s is equal to the number of occurrences of w in s, or to that number minus one if w is a suffix of s. The value

$$H_k(s) = \frac{1}{|s|} \sum_{w \in \Sigma^k} |w_s| H_0(w_s) \tag{3}$$

is called the k-th order empirical entropy of the string s. The value $|s|H_k(s)$ represents a lower bound to the compression we can achieve using codes which depend on the k most recently seen symbols. Not surprisingly, for any string s and $k \geq 0$, we have $H_{k+1}(s) \leq H_k(s)$. Note that the empirical entropy is defined for any string without any assumption on the input.

The value $H_k(s)$ is defined assuming that the first k symbols of s are coded for free. Therefore, it provides a reasonable lower bound to the compression ratio only when $|s| \gg k$. The following example shows that even when $|s| \gg k$ the empirical entropy may provide a lower bound which is too conservative and that we cannot reasonably hope to achieve.

Example 1 Let $s = \mathtt{cc}(\mathtt{ab})^n$. We have $\mathtt{a}_s = \mathtt{b}^n$, $\mathtt{b}_s = \mathtt{a}^{n-1}$, $\mathtt{c}_s = \mathtt{ca}$. This yields

$$|s|H_1(s) = nH_0(\mathtt{b}^n) + (n-1)H_0(\mathtt{a}^{n-1}) + 2H_0(\mathtt{ca}) = 2.$$

Hence, $|s|H_1(s)$ does not depend on n. ∎

The reason for which in the above example $|s|H_1(s)$ fails to provide a reasonable bound, is that for any string consisting of multiple copies of the same symbol, for example $s = a^n$, we have $H_0(s) = 0$. Since the output of any compression algorithm must contain enough information to recover the length of the input, it is natural to consider the following alternative definition of zeroth order empirical entropy. For any string s let

$$H_0^*(s) = \begin{cases} 0 & \text{if } |s| = 0, \\ (1 + \lfloor \log |s| \rfloor)/|s| & \text{if } |s| \neq 0 \text{ and } H_0(s) = 0, \\ H_0(s) & \text{otherwise.} \end{cases} \tag{4}$$

Note that $1 + \lfloor \log |s| \rfloor$ is the number of bits required to express $|s|$ in binary. Having defined H_0^* it is natural to define the k-th order modified empirical entropy H_k^* using a formula similar to (3). Unfortunately, if we simply replace H_0 with H_0^* in (3) the resulting entropy H_k^* does not satisfy the inequality $H_{k+1}^*(s) \leq H_k^*(s)$ for every string s. The reason is that, when H_0 is replaced by H_0^*, the use of a longer context for the prediction of the next symbol does not always yield an increase in compression. This observation suggests that we should define H_k^* as the maximum compression ratio we can achieve using for each symbol a codeword which depends on a context of size *at most* k (instead of always using a context of size k). We use the following notation. For a given string s, let $\mathcal{S}_k$ denote the set of all k-letter substrings of s. Let $\mathcal{Q}$ be a subset of $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_k$. We say that $\mathcal{Q}$ *covers* $\mathcal{S}_k$, and we write $\mathcal{Q} \preceq \mathcal{S}_k$, if every string $w \in \mathcal{S}_k$ has a unique suffix in $\mathcal{Q}$.

Definition 8 *For each $\mathcal{Q}$, such that $\mathcal{Q} \preceq \mathcal{S}_k$, let*

$$H_{\mathcal{Q}}^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{Q}} |w_s| H_0^*(w_s).$$

We define the k-th order modified empirical entropy *as*

$$H_k^*(s) = \min_{\mathcal{Q} \preceq \mathcal{S}_k} H_{\mathcal{Q}}^*(s). \tag{5}$$

■

The value $H_{\mathcal{Q}}^*(s)$ denotes the compression we achieve using the strings in $\mathcal{Q}$ as contexts for the prediction of the next symbol. The value $H_k^*(s)$ represents therefore the maximum compression we can achieve using contexts of length up to k.

The above discussion shows that H_k^* is a more realistic lower bound to the compression ratio we can achieve using contexts of size k or less. Since H_k has a much simpler definition and $H_k(s) \leq H_k^*(s)$ for any string s, in many situations it may be still preferable to work with H_k. However, when we are trying to establish tight bounds which hold for every string, the modified empirical entropy is more appropriate.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. *Proc. 2nd IEEE Data Compression Conference*, pages 279–288, 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*, 52(2):299–307, 1996.

[3] A. Andersson. Sorting and searching revisited. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT '96, 5th Scandinavian Workshop on Algorithm Theory*, pages 185–197. Springer-Verlag LNCS n. 1097, 3–5 July 1996.

[4] R. Baeza-Yates, A. Moffat, and G. Navarro. Searching large text collections. In J. Abello, P. Pardalos, and M. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic, 2000.

[5] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS)*, 51(1):69–82, 2000.

[6] J. Bentley. *Programming Pearls*. Addison-Wesley, USA, 1989.

[7] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive compression scheme. *Communication of the ACM*, 29(4):320–330, 1986.

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

[9] S. Chen and J. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *34th Annual Symposium on Foundations of Computer Science*, pages 104–112, Palo Alto, California, 1993. IEEE.

[10] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. of International Colloquium on Automata and Languages (ICALP '99)*, pages 261–270. Springer Verlag LNCS n. 1644, 1999.

[11] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[12] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[13] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. Assoc. Comput. Mach.*, 46:236–280, 1999.

[14] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*. ACM, 2000.

[15] D. Harman. Overview of the third text retrieval conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19. NIST Special Pubblication 500-207, 1995.

[16] H. S. Heaps. *Information retrieval: theoretical and computational aspects*. Academic Press, New York, NY, 1978.

[17] J. Kärkkäinen and E. Sutinen. Lempel-Zip index for q-grams. In J. Díaz and M. Serna, editors, *Fourth European Symposium on Algorithms (ESA '96)*, pages 378–391, Barcelona, Spain, 1996. Springer-Verlag LNCS n. 1136.

[18] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155, Recife, Brazil, 1996. Carleton University Press.

[19] A. Karlin, S. Phillips, and P. Raghavan. Markov paging (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science*, pages 208–217, Pittsburgh, Pennsylvania, 24–27 October 1992. IEEE.

[20] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[21] P. Krishnan and J. Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, December 1998.

[22] S. Kurtz. Reducing the space requirement of suffix trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[23] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[24] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, San Francisco, CA, 1994.

[25] G. Manzini. An analysis of the Burrows-Wheeler transform. Technical Report B4-99-13, IMC-CNR, 1999. `http://www.imc.pi.cnr.it/~manzini/tr-99-13/`. Preliminary version in Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA '99), 669–677.

[26] G. Manzini. The Burrows-Wheeler transform: Theory and practice. In *Proc. 24th International Symposium on Mathematical Foundations of Computer Science (MFCS '99)*, pages 34–47. Springer-Verlag LNCS n. 1672, 1999.

[27] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[28] K. Mehlhorn and M. H. Overmars. Optimal dynamization of decomposable searching problems. *Information Processing Letters*, 12(2):93–98, April 1981.

[29] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 4th South American Workshop on String Processing*. Carleton University Press, 1997.

[30] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, August 1986.

[31] J. I. Munro. Tables. In *Proceeding of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '96)*, pages 37–42. Springer-Verlag LNCS n. 1180, 1996.

[32] J. I. Munro. Succinct data structures. In *Proceeding of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '99)*. Springer-Verlag LNCS n. 1738, 1999.

[33] J. I. Munro, V. Raman, and S. Srinivasa Rao. Space efficient suffix trees. In *Proceeding of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '98)*, pages 186–195. Springer-Verlag LNCS n. 1530, 1998.

[34] G. Navarro, E. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval Journal*. To Appear.

[35] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, August 1981.

[36] K. Sadakane. A modified Burrows-Wheeler transformation for case-insensitive search with application to suffix array compression. In *DCC: Data Compression Conference*, Snowbird, Utah, 1999. IEEE Computer Society TCC.

[37] D. Salomon. *Data Compression: the Complete Reference*. Springer Verlag, 1997.

[38] J. Seward. The BZIP2 home page, 1997. `http://sourceware.cygnus.com/bzip2/index.html`.

[39] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[40] S. Wu and U. Manber. AGREP - A fast approximate pattern-matching tool. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 153–162, Berkeley, CA, USA, January 1991. Usenix Association.

[41] G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.