Opportunistic Data Structures with Applications

Paolo Ferragina[*] Giovanni Manzini[†]

Abstract

There is an upsurging interest in designing succinct data structures for basic searching problems (see [23] 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 [15], Bentley [5]). 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, though, 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 and Wheeler [7], with the structural properties of a well known indexing tool, the Suffix Array [17]. 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. More precisely, its space occupancy is optimal in an *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 uncompressible we achieve the best space bound currently known [11]; on compressible data our solution improves the succinct suffix array of [11] and the classical suffix tree and suffix array data structures either in space or in query time complexity or both.

It is a belief [27] that some space overhead should be paid to use full-text indices (like suffix trees or suffix arrays) with respect to word-based indices (like inverted lists). The results in this paper show that a full-text index may achieve *sublinear* space overhead on compressible texts. As an application we devise a variant of the well-known Glimpse tool [18] which achieves sublinear space and sublinear query time complexity. Conversely, inverted lists achieve only the second goal [27], and classical Glimpse achieves both goals but under some restrictive conditions [4].

Finally, we 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. E-mail: `ferragin@di.unipi.it`.

[†] Dipartimento di Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, Alessandria, Italy and IMC-CNR, Pisa, Italy. E-mail: `manzini@mfn.unipmn.it`.

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 [15] and Bentley [5]). This has recently motivated an upsurging interest in the design of *implicit* data structures for basic searching problems (see [23] 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. However, 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, though, 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. 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 [16, 14]), string-matching algorithms (see e.g. [1, 2, 9]), sorting and computational geometry algorithms [8].

The scenario. Most of the research in the design of indexing data structure has been mainly directed to devise solutions which offer a good trade-off between query and update time versus space usage. The two main approaches are *word-based* indices and *full-text* indices. The former achieve succinct space occupancy at the cost of being mainly limited to index linguistic texts [27], the latter achieve versatility and guaranteed performance at the cost of requiring large space occupancy (see e.g. [20, 17, 10]). Some progress on full-text indices has been recently achieved [23, 11], but an asymptotical linear space *seems* unavoidable and this makes word-based indices much more appealing when space occupancy is a primary concern. 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. 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 rather than uncompressed. Furthermore, compression may introduce some improvements which are surprisingly not confined to the space occupancy: like transfer time and, even, seek times [15, 5].

Starting from these promising considerations, many authors have recently 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 [9], LZ78 [2], Huffman [24]. All of those results, although asymptotically faster than the classical scan-based methods, they rely on the full scan of the whole *compressed* text and thus result still unacceptable for 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, achieving *experimental* trade-offs between space occupancy and query performance (see e.g. [18, 27, 4]). A significant idea towards the direct compression of the index data structure has been proposed in [12, 13] 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 [12] or worsening significantly the query performance [13].

A natural question arises at this point: Do full-text indices need a space occupancy *linear* in the (uncompressed) text size in order to support effective search operations on arbitrary patterns? It is a common belief [27] 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 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 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.[1] The data structure is provably space optimal in an *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 empirical entropy (see Appendix A). H_k expresses the maximum compression we can achieve using for each character a code which depends only on the k characters preceding it. We point out that in the case of an uncompressible string T, the space occupancy is $O(n)$ bits which is actually the best space bound currently known [11]; for a compressible string, our opportunistic data structure is the first index to achieve sublinear space occupancy. Given an arbitrary pattern $P[1, p]$, such 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 removal of the sublogarithmic term in the query bound is yet an open problem even in the uncompressed setting [11].

The novelty of our approach resides in the careful combination of the Burrows-Wheeler compression algorithm [7] with the the Suffix Array data structure [17] to obtain a sort of *compressed* suffix array. We indeed show how to *augment* the information kept by the Burrows-Wheeler algorithm, 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. The first algorithm is 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 3.1); the second algorithm exploits compression to speed up the retrieval of the actual positions of the pattern occurrences, thus incurring only in a *sublogarithmic* $O(\log^{\epsilon} u)$ time slowdown (for any fixed $\epsilon > 0$, see Section 3.2).

In some sense, our result can be interpreted as a method to *compress* the suffix array, and still support effective searches for arbitrary patterns. In their seminal paper, Manber and Myers [17] 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 $\Theta(u \log u)$ bits of storage. Recently, Grossi and Vitter [11] reduced the space usage of suffix arrays to $\Theta(u)$ bits at the cost of requiring $O(\log^{\epsilon} u)$ time to retrieve the i-th suffix. Hence, searching in this succinct suffix array via the classical Manber-Myers' procedure takes $O(p + \log^{1+\epsilon} u + occ \log^{\epsilon} u)$ time. Our solution therefore improves the succinct suffix array of [11] both in space and query time complexity. The authors of [11] introduce also other hybrid indices based on the combination of their succinct suffix array and various known techniques; they achieve $O(\frac{p}{\log u} + occ \log^{\epsilon} u)$ query-time complexity but yet require $\Omega(u)$ bits of storage. 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 $\Omega(p)$ time complexity and occupy $\Omega(u \log u)$ bits.

In Section 4, we investigate the modifiability of our opportunistic data structure by studying how to choreograph its basic ideas with a dynamic setting, and 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, 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. We point out that even in the case of an uncompressible text T, our space bounds are the best known ones since the data structures in [11] do not support updates (the dynamic case is left as open in their Section 4).

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 [18]). The notable feature

[1] The concept of *opportunistic algorithm* has been introduced in [9] 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*.

of block addressing is that it can achieve both sublinear space overhead and sublinear query time, whereas inverted indices achieve only the second goal [4]. Unfortunately, up to now all the known block addressing indices [18, 4] achieve this goal only under some restrictive conditions on 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

Let $T[1,u]$ be a text drawn from a constant-size alphabet Σ. A central concept in our discussion is the *suffix array* data structure [17]. The suffix array $\mathcal{A}$ built on $T[1,u]$ 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 $\mathcal{A} = [1,3,2,4,5]$. In practice $\mathcal{A}$ occupies $4u$ bytes, actually a lot when indexing large text collections. It is a long standing belief that suffix arrays are uncompressible because of the "apparently random" permutation of the suffix pointers. Recent results in the data compression field have opened the door to revolutionary ways to compress suffix arrays and are the basic tools of our solution. In [7], Burrows and Wheeler proposed a transformation (BWT from now on) consisting of a reversible permutation of the text characters which gives a new string that is "easier to compress". The BWT tends to group together characters which occur adjacent to similar text substrings. This nice property is exploited by locally-adaptive compression algorithms, such as move-to-front coding [6], in combination with statistical (i.e. Huffman or Arithmetic coders) or structured coding models. The BWT-based compressors are among the best compressors currently available since they achieve a very good compression ratio using relatively small resources (time and space).

The reversible BW-transform. We distinguish between a *forward* transformation, which produces the string to be compressed, and a *backward* transformation which gives back the original text from the transformed one. The forward BWT consists of three basic steps: (1) Append to the end of T a special character # smaller than any other text character; (2) form a *conceptual* 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 every column of $\mathcal{M}$ is a permutation of the last column L, and in particular the first column of $\mathcal{M}$, call it F, is obtained by lexicographically sorting the characters in L.

There is a strong apparent relation between the matrix $\mathcal{M}$ and the suffix array $\mathcal{A}$ of the string $T\#$. When sorting the rows of the matrix $\mathcal{M}$ we are essentially sorting the suffixes of $T\#$. Consequently, entry $\mathcal{A}[i]$ points to the suffix of $T\#$ occupying (a prefix of) the ith row of $\mathcal{M}$. The cost of performing the forward BWT is given by the cost of constructing the suffix array $\mathcal{A}$, and this requires $O(u)$ time [20].

The cyclic shift of the rows of $\mathcal{M}$ is crucial to define the backward BWT, which is based on two easy to prove observations [7]:

- **a.** Given the ith row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T, namely $T = \cdots L[i]F[i] \cdots$.
- **b.** Let $L[i] = c$ and let r_i be the rank of the row $\mathcal{M}[i]$ among all the rows *ending* with the character c. Take the row $\mathcal{M}[j]$ as the r_i-th row of $\mathcal{M}$ *starting* with c. Then the character corresponding to $L[i]$ in the first column F is located at $F[j]$ (we call this *LF-mapping*, where $LF[i] = j$).

We are therefore ready to describe the backward BWT:

1. Compute the array $C[1 \dots |\Sigma|]$ storing in $C[c]$ the number of occurrences of characters $\{\#, 1, \dots, c-1\}$ in the text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).
2. Define the LF-mapping $LF[1 \dots u+1]$ as follows $LF[i] = C[L[i]] + r_i$, where r_i equals the number of occurrences of character $L[i]$ in the prefix $L[1,i]$ (see observation (b) above).
3. Reconstruct T backward as follows: set $s = 1$ and $T[u] = L[1]$ (because $\mathcal{M}[1] = \#T$); then, for each $i = u-1, \dots, 1$ do $s = LF[s]$ and $T[i] = L[s]$.

In [26] it has been shown how to derive the suffix array $\mathcal{A}$ from L in linear time; however in the context of pattern searching this approach is no better than the known scan-based opportunistic algorithms [9]. Nonetheless, the *implicit* presence of the suffix array $\mathcal{A}$ into L stimulates the investigation of the possibility to take full advantage of the structure of $\mathcal{A}$ for fast searching, and of the succinct compression of L for reducing the overall space occupancy. This is actually the ultimate hope of any indexer: succinct and fast! In the next section, we show that this result is achievable provided that a sublogarithmic slowdown is introduced in the cost of listing all pattern occurrences.

In order to describe our results on searching in BWT-compressed texts, we must commit ourselves to one of several algorithms based on the BWT. Let $T^{bw} = \texttt{bwt}(T)$ denote the last column L, output of the BW-transform. We compress T^{bw} in three steps (see also [19][2]):

1. Use a move-to-front coder, briefly `mtf` [6], to encode a character c via the count of distinct characters seen since its previous occurrence. The structural properties of T^{bw}, mentioned above, imply that the string $T^{mtf} = \texttt{mtf}(T^{bw})$ will be dominated by *low numbers.*
2. Encode each run of zeroes in T^{mtf} using run length encoding (`rle`). More precisely, replace the sequence 0^m with the number $(m+1)$ written in binary, least significant bit first, discarding the most significant bit. For this encoding we use two new symbols $\mathbf{0}$ and $\mathbf{1}$ so that the resulting string $T^{rl} = \texttt{rle}(T^{mtf})$ is over the alphabet $\{\mathbf{0}, \mathbf{1}, 1, 2, \ldots, |\Sigma| - 1\}$.
3. Compress T^{rl} by means of a variable-length prefix code, called `PC`, which encodes the symbols $\mathbf{0}$ and $\mathbf{1}$ using two bits (10 for $\mathbf{0}$, 11 for $\mathbf{1}$), and the symbol i using a variable-length prefix code of $1 + 2\lfloor \log(i+1) \rfloor$ bits, where $i = 1, 2, \ldots, |\Sigma| - 1$.

The resulting algorithm $\texttt{BW\_RLX} = \texttt{bwt} + \texttt{mtf} + \texttt{rle} + \texttt{PC}$ is sufficiently simple so that in the rest of the paper we can concentrate on the searching algorithm without being distracted by the details of the compression. Despite of the simplicity of `BW_RLX`, using the results in [19] it is possible to show that (proof in the full paper), for any $k \geq 0$ there exists a constant g_k such that

$$|\texttt{BW\_RLX}(T)| \leq 5\ |T|\ H_k(T) + g_k \log |T| \qquad (1)$$

where H_k is the kth order empirical entropy. H_k expresses the maximum compression we can achieve using for each character a code which depends only on the k characters preceding it. The bound (1) tells us that even for strings with very small entropy `BW_RLX`'s output size is within a constant factor of the optimum. Note that such a similar bound has not been proven for any LZ-based algorithm [19].

3 Searching in BWT-compressed text

Let $T[1, u]$ denote an arbitrary text over the alphabet Σ, and let $Z = \texttt{BW\_RLX}(T)$. 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 above). Recall that the suffix array $\mathcal{A}$ posses two nice structural properties which are usually exploited to support fast pattern searches: (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.

3.1 First step: Counting the occurrences

We now describe an algorithm, called BW_Search, which identifies the positions *sp* and *ep* by accessing only the compressed string Z and some auxiliary *array-based* data structures.

[2] The algorithm considered in this paper corresponds to the procedure `A*` described in [19]

```
Algorithm 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.         $c = P[i-1]$;
 5.         $sp = C[c] + \mathsf{Occ}(c, 1, sp-1) + 1$;
 6.         $ep = C[c] + \mathsf{Occ}(c, 1, ep)$;
 7.         $i = i - 1$;
 8. if ($ep < sp$) then return "not found" else return "found ($ep - sp + 1$) occurrences".
```

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

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. 1. We start with $i = p$ (step 1), so that sp and ep can be determined via the array C defined in Section 2 (step 2). If $ep < sp$ then we can conclude that $P[p]$ does not occur in T and hence P does not too. The subroutine $\mathsf{Occ}(c, 1, k)$ used in Steps 5 and 6 reports the number of occurrences of the character c in $T^{bw}[1,k]$; hence it is employed to implement the LF-mapping. 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. The following lemma proves the correctness of BW_Search, assuming Occ works as claimed (proof in Appendix B).

Lemma 1 *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 $P[i-1,p]$.*

The running time of BW_Search depends on the cost of procedure Occ. We now describe an algorithm for computing $\mathsf{Occ}(c, 1, k)$ in $O(1)$ time, 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 $BT_1^{mtf}, \ldots, BT_{u/\ell}^{mtf}$ of size ℓ too. 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. The general case in which a sequence of zeroes may span several buckets is discussed in Appendix C. Under our assumption, the buckets BT_i^{mtf}'s induce a partition of the compressed file Z into u/ℓ *compressed buckets* $BZ_1, \ldots, BZ_{u/\ell}$, defined as $BZ_i = \mathtt{PC}(\mathtt{rle}(BT_i^{mtf}))$.

Let BT_i denote the bucket containing the character $T^{bw}[k]$ ($i = \lceil k/\ell \rceil$). The computation of $\mathsf{Occ}(c, 1, k)$ is based on a hierarchical decomposition of $T^{bw}[1,k]$ in three substrings as follows: (i) the longest prefix of $T^{bw}[1,k]$ having length a multiple of ℓ^2 (i.e. $BT_1 \cdots BT_{\ell i^*}$, 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. $BT_{\ell i^* + 1} \cdots BT_{i-1}$), and finally (iii) the remaining suffix of $T^{bw}[1,k]$ which is indeed a prefix of the bucket BT_i. We compute $\mathsf{Occ}(c, 1, k)$ by summing the number of occurrences of c in each of these substrings. This can be done in $O(1)$ time and sublinear space using the following auxiliary data structures:

- For the calculations on the substring of point (i):
  - For $i = 1, \ldots, u/\ell^2$, the array $NO_i[1, |\Sigma|]$ stores in entry $NO_i[c]$ the number of occurrences of character c in $BT_1 \cdots BT_{i\ell}$.

  - The array $W[1, u/\ell^2]$ stores in entry $W[i]$ the value $\sum_{j=1}^{i\ell} |BZ_j|$ equals to the sum of the sizes of the compressed buckets $BZ_1, \ldots, BZ_{i\ell}$.

- For the calculations on the substring of point (ii):
  - For $i = 1, \ldots, u/\ell$, the array $NO'_i[1, |\Sigma|]$ stores in entry $NO'_i[c]$ the number of occurrences of character c in the string $BT_{i^*+1} \cdots BT_{i-1}$ (this concatenated string has length less than ℓ^2).
  - The array $W'[1, u/\ell]$ stores in entry $W'[i]$ the value $\sum_{j=i^*+1}^{i-1} |BZ_j|$ equals to 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:
  - The array $MTF[1, u/\ell]$ stores in 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 (i.e. $O(1)$ 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 in $O(1)$ time. Table S has $O(\ell 2^{\ell'})$ entries each one occupying $\log \ell$ bits.

The computation of $\mathsf{Occ}(c, 1, k)$ proceeds as follows. First, the bucket BT_i containing the character $c = T^{bw}[k]$ is determined via $i = \lceil k/\ell \rceil$, together with the position $j = k - (i-1)\ell$ of this character in BT_i and the parameter $i^* = \lfloor (k-1)/\ell^2 \rfloor$. Then the number of occurrences of c in the prefix $BT_1 \cdots BT_{\ell i^*}$ (point (i)) is determined via $NO_{i^*}[c]$, and the number of occurrences of c in the substring $BT_{\ell i^*}, \ldots, BT_{i-1}$ (point (ii)) is determined via $NO'_i[c]$. Finally, the compressed bucket BZ_i is retrieved from Z (notice that $W[i^*] + W'[i] + 1$ is its starting position), and the number of occurrences of c within $BT_i[1, j]$ are accounted using table S. The sum of these three quantities gives $\mathsf{Occ}(c, 1, k)$.

By construction any compressed bucket BZ_i has size at most $\ell' = (1 + 2\lfloor \log \Sigma \rfloor)\ell$ bits. Now we set $\ell = \Theta(\log u)$ 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. 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 can now state 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 $|Z| + O\Big(\frac{u}{\log u} \log\log u\Big)$ bits.*

3.2 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: the first one is simple but slow, the second one is faster and relies on the very special properties of the the string T^{bw} and on a different compression algorithm.

In the first algorithm we *logically mark* the rows of $\mathcal{M}$ which correspond to text positions having the form $1 + i\eta$, for $\eta = \Theta(\log^2 u)$ and $i = 0, 1, \ldots, u/\eta$. We store with these marked rows the starting positions of the corresponding text suffixes explicitly. This preprocessing is done at compression time. At query time we find $pos(s)$ as follows. 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 $T[pos(s) - 1, u]$. We iterate this procedure v times until s' points to a marked row; then we compute $pos(s) = pos(s') + v$. The crucial point of the algorithm is the logical marking of the rows of $\mathcal{M}$ corresponding to the text suffixes starting at positions $1 + i\eta$, $i = 0, \dots, u/\eta$. Our solution consists in storing the row numbers in a two-level bucketing scheme. 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 $O(\log\log u)$ bits long, membership queries take $O(1)$ time on a RAM. The overall space required for the logical marking is $O((u/\eta)\log\log u)$ bits. In addition, 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 occupancy is $O((u/\eta)\log u) = O(u/\log u)$ bits. For what concerns the time complexity, our algorithm computes $pos(s)$ in at most $\eta = \Theta(\log^2 u)$ steps, each taking constant time. Hence the *occ* occurrences of a pattern P in T can be retrieved in $O(occ\ \log^2 p)$ time, with a space overhead of $O(u/\log u)$ bits. Combining the results of this section with equation (1):

Theorem 2 *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 $k \geq 0$.* ■

We now refine the above algorithm in order to compute *pos*(s) in $O(\log^\epsilon u)$ time, for any fixed $\epsilon > 0$. We still use the idea of marking some of the rows in $\mathcal{M}$, however we introduce some *shortcuts* which allow to move in T by more that one character at a time, thus reducing the number of steps required to reach a marked position. The key ingredient of our new approach is a procedure for computing the LF-mapping over a string $\bar{T}$ drawn from an alphabet Λ of *non-constant* size. The procedure is based on an alternative compression of the output $\bar{T}^{bw}$. We make use of the following lemma (proof and details are given Appendix D):

Lemma 2 *Given a string $\bar{T}[1, v]$ over an arbitrary alphabet Λ, we can compute the LF-mapping over $\bar{T}^{bw}$ in $O(\log^\epsilon v)$ time using $O\Big(v(1 + H_k(\bar{T})) + |\Lambda|^{k+1}(\log|\Lambda| + \log v)\Big)$ bits of storage, for any given $\epsilon > 0$.*

We now show how to use Lemma 2 to compute *pos*(s) in $O(\log^{(1/2)+2\epsilon} u)$ time; this is an intermediate result that will be then refined to achieve the final $O(\log^\epsilon u)$ time bound.

At compression time we logically mark the rows of $\mathcal{M}$ which correspond to text positions of the form $1 + i\gamma$ for $i = 0, \dots, u/\gamma$ and $\gamma = \Theta(\log^{(1/2)+\epsilon} u)$. Then, we consider the string T_0 obtained by grouping the characters of T into 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. It is easy to see that $\mathcal{M}_0$ consists of the marked rows of $\mathcal{M}$. Now we mark the rows of $\mathcal{M}_0$ corresponding to the suffixes of T_0 starting at the positions $1 + i\eta$, for $i = 0, \dots, |T_0|/\eta$ and $\eta = \Theta(\log^{(1/2)+\epsilon} u)$. For these rows we explicitly keep the starting position of the corresponding text suffixes. To compute *pos*(s) we first compute the LF-mapping in $\mathcal{M}$ until we reach a marked row s'. Then we compute *pos*(s') by finding its corresponding row in $\mathcal{M}_0$ and computing the LF-mapping in $\mathcal{M}_0$ (via Lemma 2) until we reach a marked row s'' in $\mathcal{M}_0$ (for which *pos*(s'') is explicitly available by construction). 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 using again a Packed B-tree and a two level bucketing scheme as before. In addition, for $\Theta(|T_0|/\eta)$ rows of $\mathcal{M}_0$ we keep explicitly their positions in T_0 which take $\Theta((|T_0|/\eta)\log u) = \Theta(u/\log^{2\epsilon} u)$ bits of storage. The space occupancy of the procedure for computing the LF-mapping in T_0^{bw} is given by Lemma 2. Since $H_k(T_0) \leq \gamma\ H_{k\gamma}(T)$, 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 complexity of the algorithm is $O(\gamma)$ (for finding a marked row in $\mathcal{M}$) plus $O(\eta\log^\epsilon u)$ (for finding a marked row in $\mathcal{M}_0$), thus $O(\log^{(1/2)+2\epsilon} u)$ time overall.

The final time bound of $O(\log^\epsilon u)$ for the computation of $pos(s)$ can be achieved by iterating the approach above as follows. The main idea is to take $\gamma_0 = \Theta(\log^\epsilon u)$, and apply the procedure for computing the LF-mapping in T_0 for $\Theta(\log^\epsilon u)$ steps, thus identifying a row s_1 of the matrix $\mathcal{M}_0$ such that[3] $pos(s_1)$ has the form $1+i\gamma_1$ with $\gamma_1 = \Theta(\log^{2\epsilon} u)$. Next, we define the string T_1 obtained by grouping the characters of T into blocks of size γ_1 and we consider the corresponding matrix $\mathcal{M}_1$. By construction s_1 corresponds to a row in $\mathcal{M}_1$ and we can iterate the above scheme. At the jth step we operate on the matrix $\mathcal{M}_{j-1}$ until we find a row s_j such that $pos(s_j)$ has the form $1 + i\gamma_j$ where $\gamma_j = \Theta(\log^{(j+1)\epsilon} u)$. This continues until j reaches the value $\lceil 1/\epsilon \rceil$. At that point the matrix $\mathcal{M}_j$ consists of $\Theta(u/\log^{1+\delta} u)$ rows and thus we can store explicitly the starting position of their corresponding text suffixes. Summing up, the algorithm computes $pos(s)$ in $1/\epsilon = \Theta(1)$ iterations, each taking $\Theta(\log^{2\epsilon} u)$ time. Since ϵ is an arbitrary positive constant, it is clear that we can rewrite the previous time bound as $\Theta(\log^\epsilon u)$. The space occupancy is dominated by the one required for the marking of $\mathcal{M}$.

Theorem 3 *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 $k \geq 0$.*

4 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 support fast searches for the occurrences of an arbitrary pattern $P[1,p]$ into Δ's texts. This problem can be solved in optimal time complexity and $\Theta(u \log u)$ bits of storage [20, 10]. In the present section we aim at *dynamizing* our compressed index in order to keep Δ in a reduced space and be able to efficiently support update and search operations. Our result exploits an elegant technique proposed at the beginning of '80 in [25, 21], here adapted to manage items (i.e. texts) of variable lengths.

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 be possibly 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.

In the following we 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 $O(p \log^3 u + occ)$ 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. Searching for an arbitrary pattern $P[1,p]$ in Δ, with $p = O(\frac{u}{\log^2 u})$ 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.

[3] In this paragraph pos always refer to the starting position in the text T of the suffix corresponding to a row.

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, the 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 not empty 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. Some care must be taken to evaluate the space occupied during the reconstruction of the set $\mathcal{S}_i^h$. 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.

We now show how to support text deletions from Δ. The main problem here is that from one side we would like to physically cancel the 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 the $\mathcal{S}_i^h$'s. For the sake of presentation let T^{bw} be the BWT of the texts stored in some set $\mathcal{S}_i^h$. We store in a balanced search tree the set $\mathcal{I}_i^h$ of interval positions in T^{bw} occupied by deleted text suffixes. If a pattern occurrence is found in T^{bw} using our compressed index, we can check in $O(\log u)$ time if it is a real or a ghost occurrence. Every time a text $T[1,t]$ must be deleted from $\mathcal{S}_i^h$, we search for all of its suffixes in $\mathcal{S}_i^h$ 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 $\mathcal{S}_i^h$ as soon as a fraction of $\Theta(\frac{1}{\log^2 u})$ suffixes is 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 $\mathcal{S}_{i-1}^h$. Consequently the amortized cost of delete is $O(t \log u + t \log^2 u) = O(t \log^2 u)$, where the first 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.

Finally, to identify a text to be deleted we append to every text in Δ an identifier of $O(\log u)$ bits, and we keep track of the subset $\mathcal{S}_i^h$ containing a text via a table. This introduces an overhead of $O(m \log u)$ bits which is $o(u)$ if we reasonably assume that the texts are not too short, i.e. $\omega(\log u)$ bits each.

Theorem 4 *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 $k \geq 0$.*

5 A simple application

Glimpse [18] is an effective tool to index linguistic texts. From a high level point of view, it 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: pointers to word occurrences are shorter, and the occurrences of the same word in a single block are represented only once. Typically the index is very compact, 2–4% of the original text size [18].

Given this index structure, the search scheme proceeds in two steps: first the queried word w is searched in V, then all candidate blocks of $L(w)$ 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* [28] both in the vocabulary and in the block searches. Clearly, the search is efficient if the vocabulary is small, if the query is enough selective, and if the block size is not too large. 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 scheme [18, 4] have shown that the Glimpse approach is effective only for medium sized texts (roughly up to 200Mb). Recent papers tried to overcome this limitation by compressing each text block individually and then searching it via proper opportunistic string-matching algorithms [18, 24]. The experimental results showed an improvement of about 30–50% in the final performance, thus implicitly proving that the second searching step dominates Glimpse's query performance.

Our opportunistic index naturally fits in this block-addressing framework and allows us to extend its applicability to larger text databases. The new approach, named *Compressed Glimpse* (shortly `CGlimpse`), consists of using our opportunistic data structure to index each text block individually; this way, each candidate block is not fully scanned at query time but its index is employed to fasten the detection of the pattern occurrences. 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 feasible using a model generally accepted in Information Retrieval [4]. It assumes the Heaps law to model the vocabulary size (i.e. $V = O(u^\beta)$ with $0 < \beta < 1$), the generalized Zipf's law to model the frequency of words in the text collection (i.e. the largest ith frequency of a word is $u/(i^\theta H_V^{(\theta)})$, where $H_V^{(\theta)}$ is a normalization term and θ is a parameter larger than 1), and assumes that $O(u^\rho)$ is the number of matches for a given word with $k \geq 1$ errors (where $\rho < 1$). Under these hypothesis we can show that `CGlimpse` achieves *both sublinear space overhead and sublinear query time independent of the block size* (proof in the full paper). Conversely, inverted indices achieve only the second goal [27], and classical Glimpse achieves both goals but under some restrictive conditions on the block size [4].

6 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. This is an open problem also in the uncompressed setting [11]. In external memory, 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 listing of the pattern occurrences by a factor $\Theta(B)$. As far as the problem of counting the number of pattern occurrences is concerned, we are investigating the use of the technique in Lemma 2 to achieve $o(p)$ time bound.

Another interesting issue concerns with the design of word-based indices where the searches are mainly confined to words or prefixes of words. We are investigating a novel word-based compression scheme which compresses the pattern before searching it into the opportunistic data structure thus achieving $o(p)$ I/O-complexity.

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. rishnan 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.

[14] 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.

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

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

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

[18] 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.

[19] G. Manzini. An analysis of the Burrows-Wheeler transform. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 669–677. Full version in Tech. Rep. B4-99-13, IMC-CNR, 1999. `http://www.imc.pi.cnr.it/~manzini/tr-99-13/`.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

A The empirical entropy of a string

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.

B Proof of Lemma 1

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 the ordering of the rows in $\mathcal{M}$ (see Section 2), s' must be the first row 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 the properties of LF-mapping (see Section 2), 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).

C Managing long runs of zeroes

In this appendix we discuss the computation of $\mathsf{Occ}(c, 1, k)$ in the general case in which T^{mtf} contains one or more runs of zeroes which cross the border of the buckets BT_i^{mtf}'s. Recall that `rle` replaces each (maximal) run of zeroes 0^m in T^{mtf} with the string $Bin(m)$ defined as the binary representation of $(m+1)$ in reverse order (least significant bit first) and with the most significant bit discarded. This encoding is also known as `1/2`-encoding since if $Bin(m) = b_0 b_1 \cdots b_k$ then $m = \sum_{j=0}^{k}(b_j+1)2^j$. Because of this property, from $Bin(m)$ we can retrieve 0^m by simply replacing each bit b_j with a sequence of $(b_j+1)2^j$ zeroes.

Suppose now 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 sequence overlapping an entire bucket). We define the compressed buckets BZ_{i-1} and BZ_i as follows. We know 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 mentioned above 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 S introduced in Section 3.2 can be used to count the number of occurrences of a character in a prefix of BZ_{i-1} but does not work for BZ_i. Indeed, 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 $LT[i] = b$). Figure 2 shows the resulting algorithm for the computation of the number of occurrences of c among the first j characters of a compressed bucket BZ_i.

```
if $c$ is the first character in $MTF[i]$ then
    if $j \leq LT[i]$ then return $j$
    else return $LT[i] + S[c, j - LT[i], Z[s_i + 1 + LZ[i], s_i + \ell'], MTF[i]]$
else
    if $j \leq LT[i]$ then return $0$
    else return $S[c, j - LT[i], Z[s_i + 1 + LZ[i], s_i + \ell'], MTF[i]]$
```

Figure 2: Computation of the number of occurrences of c among the first j characters of BZ_i.

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 procedure Occ.

D Proof of Lemma 2

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

Now consider a modified move-to-front procedure, called $\texttt{mtf}^*$, that encodes a character in $\bar{T}^{bw}$ by the *number of characters seen* since its previous occurrence[4]. For $j = 0, 1, \ldots, 2/\epsilon$, let $\bar{T}^{mtf^*}_j = \texttt{mtf}^*(\bar{T}^{bw}_j)$. We store the strings $\bar{T}^{mtf^*}_0, \bar{T}^{mtf^*}_1, \ldots, \bar{T}^{mtf^*}_{2/\epsilon}$ and, for the characters in $\bar{T}^{bw}_{2/\epsilon}$, 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 $\bar{T}^{mtf^*}_j$'s to jump, in $\Theta((2/\epsilon)\log^\epsilon v) = \Theta(\log^\epsilon v)$ steps, to an occurrence of α which is also in $\bar{T}^{bw}_{2/\epsilon}$. For this occurrence we have the explicit LF-mapping and we can easily reconstruct $LF[s]$. More precisely, we start from $\bar{T}^{bw}_0[s]$ and we move to $\bar{T}^{bw}_1, \bar{T}^{bw}_2, \bar{T}^{bw}_{2/\epsilon}$ updating a counter v as follows. Let $\bar{T}^{bw}_j[i]$ 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 $\bar{T}^{bw}_j$ using the distance encoded in $\bar{T}^{mtf^*}_j[i]$ and sum $\log^{j\,\epsilon} v$ to the counter v. Note that we are indeed implicitly jumping $\log^{j\,\epsilon} v$ 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 $\bar{T}^{bw}_{j+1}$. After $\Theta(\log^\epsilon v)$ iterations, we reach a copy of α in $\bar{T}^{bw}_{2/\epsilon}$ for which its LF-mapping over $\bar{T}^{bw}$ is available. Consequently, we sum this value with v and thus obtain $LF[s]$ in overall $\Theta(\log^{2\epsilon} v)$ time.

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

- Z_j is a string succinctly encoding $\bar{T}^{mtf^*}_j$ and consisting of the concatenation $\texttt{bin}(\bar{T}^{mtf^*}_j[1])\texttt{bin}(\bar{T}^{mtf^*}_j[2]) \cdots \texttt{bin}(\bar{T}^{mtf^*}_j[v/\log^{j\,\epsilon} v])$;

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

- 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 [22] and applied on the arrays above, we can implement each step of the computation of $LF[s]$ described before in $O(1)$ time. Namely, 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 this proves the time bound of Lemma 2.

To prove the space bound we first observe that the space occupancy induced by the explicit LF-mapping kept for the characters in $\bar{T}_{2/\epsilon}^{bw}$ is $O(v/\log v)$ bits (we store $O(v/\log^2 v)$ entries of $O(\log v)$ bits each). The arrays Z_j's, W_j's, N_j's and the data structures for the `rank` and `select` operations take $O(|Z_0| + |Z_1| + \cdots + |Z_{2/\epsilon}|)$ bits of storage, hence $O(|Z_0|)$ bits. The lemma follows since from [6, 19] we can derive the relation $|Z_0| = O(v(1 + H_k(\bar{T})) + |\Lambda|^{k+1}(\log|\Lambda| + \log v))$.