Noname manuscript No.
(will be inserted by the editor)

A Compressed Self-Index using a Ziv-Lempel Dictionary

Luís M. S. Russo*, **Arlindo L. Oliveira**

INESC-ID/IST
e-mail: {lsr,aml}@algos.inesc-id.pt

The date of receipt and acceptance will be inserted by the editor

Abstract A compressed full-text self-index for a text T, of size u, is a data structure used to search for patterns P, of size m, in T, that requires reduced space, i.e. space that depends on the empirical entropy (H_k or H_0) of T, and is, furthermore, able to reproduce any substring of T. In this paper we present a new compressed self-index able to locate the occurrences of P in $O((m + occ)\log u)$ time, where occ is the number of occurrences. The fundamental improvement over previous LZ78 based indexes is the reduction of the search time dependency on m from $O(m^2)$ to $O(m)$. To achieve this result we point out the main obstacle to linear time algorithms based on LZ78 data compression and expose and explore the nature of a recurrent structure in LZ-indexes, the $\mathcal{T}_{78}$ suffix tree. We show that our method is very competitive in practice by comparing it against other state of the art compressed indexes.

1 Introduction and related work

The exact matching problem consists in searching for a short sequence P (the pattern) in a longer sequence T (the text). Naive and linear time solutions for this problem can be found in undergraduate computer science textbooks [4]. This problem has outgrown its initial motivation, text editing subroutines. Text databases storing large amounts of information such as pitch sequences, DNA or protein sequences, large natural texts, program code, etc, need fast pattern matching algorithms. With the increasing amount of digital information available, on-line approaches to the problem are no longer viable. The study of index data structures, that are able to reduce the time it takes to locate the occurrences of P, has been the focus of the string processing community for several years. Classical indexes, however, have a tendency to be space intensive. This constitutes a severe problem, since not being able to store indexes in main memory limits their usage.

In recent years a new and extremely successful approach to this problem has emerged. *Compressed full-text indexes*, which use data compression techniques to produce data structures that are less space demanding have been proposed by several researchers [5, 10, 13, 21, 27]. Compressed indexes consist of a careful combination of *text compression* and *succinct data structures* with indexing data structures. Navarro and Mäkinen presented a comprehensive survey on compressed full-text indexes [20].

A *text compression* technique is a way to encode the text in a format that requires less space than that of the original raw sequence and that still represents the original text. By representation we mean that we can consult any part of the original text, even if this implies that first we

* Supported by the Portuguese Science and Technology Foundation by grant SFRH/BD/12101/2003 in project POCI 2010 and Project BIOGRID POSI/SRI/47778/2002

decompress the whole string. The idea is that an index based on the compressed format may also require less space. In fact, it turns out that data compression algorithms explore the internal structure of a string much in the same way that indexes do. It should be clear that we wish to recover exactly the original text, i.e. we are interested only in "lossless" data compression methods. Text compression therefore provides a trade-off between the size necessary to store the text and the time it takes to consult a part of the text. This trade-off might be advantageous for storing a text or for transmitting it, such as over the Internet, from secondary memory to main memory or from main memory to cache. Storing compressed files saves storage space. Transmitting compressed files saves time when the overall time to encode, transmit and decode the file is smaller than the time to transmit the original text. Therefore applications such as gzip or bzip2 became popular for compressing and decompressing texts.

Text compression cannot compress a string by an arbitrary amount. In fact a simple argument proves that even if we had enough computational power available, it is not possible to compress every text by 1 bit. A lower bound on how much a string generated by a given source can be compressed was given by Shannon. In Shannon's theory different strings are grouped together into ergodic sources. Observe that for every text, individually, it is possible to find a program that outputs it. To avoid this pathological solution we can use Kolmogorov complexity which considers the size of the program that generates the string as part of the complexity of the string. The fundamental problem with Kolmogorov complexity is that it is not computable. In this work we use a more pragmatic approach. We do not wish to make any assumptions on how the text was generated. Moreover we are not so much interested in how much a text can be compressed in theory as we are in how much it can be compressed by a class of "good" compressors. We will use the notion of *k-th order empirical entropy* $H_k(T)$ given by Manzini [16]. The *k-th order empirical entropy* gives a lower bound on the best compression ratio that can be applied to T, if, when compressing a character of T, we consider only the context of the k characters that precede it in T. Obviously the larger the context we consider, the better the compression should be, i.e. $0 \leq H_k(T) \leq \ldots \leq H_0(T) \leq \log \sigma$ (where by log we mean $\log_2$). Therefore the size of the compressed text will range from $uH_k(T)$ to $uH_0(T)$ depending on the compressor we use. Moreover, empirical entropy provides a measure of the complexity of T taken as a finite object. This is opposed to the classical notion of entropy by Shannon. State of the art compressed indexes consider T as finite and organize it globally. In a way, our contribution is to organize globally Ziv-Lempel compressed indexes that were only locally organized.

A *succinct data structure* representation of a data structure is a compact representation of it. Trees are a recurrent data structure in computer science and, in particular, play a central role in *full-text indexing* theory. It is therefore natural to consider succinct representations of trees. Clearly the less space we need to represent a tree, the less space our indexes will require. Jacobson [12] was the first to study succinct data structures, such as trees and bitmaps (strings of 1's and 0's). Trees are commonly implemented with pointers which may not be the most space efficient way to store them. A tree, can, for example, be represented as a string of left and right parentheses. This representation does not support by itself common operations efficiently, such as moving to a father node or to a child node, but it does represent the tree. Therefore, a tree with n nodes can be represented with $2n$ bits. The work presented by Jacobson showed how to simulate tree traversals efficiently using only $o(n)$ extra bits. Clearly this kind of results is relevant for producing smaller *full-text indexes*.

The fundamental tools supporting these kinds of data structures are the Rank and Select operations over bitmaps. The Rank operation counts the number of 1's up to a given position in the bitmap. The Select operation returns the location of the i-th 1 in the bitmap. Jacobson showed how to compute Rank in constant time, with only $o(n)$ extra bits. Later on, Munro and Clark [17] obtained constant-time solutions for Select, with $o(n)$ extra bits. The set of operations provided by succinct trees has been successively enlarged and improved by several researchers; including Munro et al. [19], Benoit et al. [2] and Geary et al. [7]. The Rank and Select operations also proved to be useful for representing permutations [18]. Trees and permutations play a central role in *full-text indexing* theory. Hence, this kind of results account for a significant part of the success of *compressed indexes*.

Producing compressed indexes lead to new discoveries about full-text indexes. A surprising such discovery was self-indexing. Basically it turned out that with a negligible amount of information, it is possible to make full-text indexes capable of reproducing any substring of T without storing T explicitly. Another important discovery is backward searching, which is the operating principle behind the FM-Index [5].

Compressed suffix arrays [10, 27] and the FM-index [5] are the main trends of compressed indexes. This is partially due to the fact that LZ-indexes [5, 13, 21] require a considerable amount of time to determine the number of occurrences of P in T, denoted by occ. In fact, the index of Kärkkäinen et al. [13], which was not a self-index, required $O(m^2 + (m + occ) \log u)$ time and Navarro's [21] index required $O((m^3 \log \sigma) + (m + occ) \log u)$ which was recently improved to $O((m^2 \log m) + (m + occ) \log u)$ by Arroyuelo et al. [1]. It can be seen that in all these approaches the dependency on m is at least $O(m^2)$. The only LZ based index that was able to achieve $O(m)$ time was presented by Ferragina et al. [5]. However, this index requires a considerable amount of space, $O(uH_k(T) \log^{\epsilon} u) + o(u)$ bits, ignoring the dependency on σ. In fact, the index presented by Ferragina et al. has not been implemented. The structure we propose is very similar to the one given by Ferragina et al. In fact we use essentially the same structures they do. However the operations permitted and the representation used are new. If they used the same range data structure we use, their structure would not have a $\log^{\epsilon} u$ dependency on the space complexity. However, since their approach is heavily dependent on the FM-Index, it may lead to alphabet related problems, i.e. large hidden σ dependencies. This problem, however, has been recently addressed [6, 8] and is, therefore, solvable. Nevertheless our approach is simpler and alphabet independent.

The Ziv-Lempel algorithm is a dictionary based compression method. In essence, the idea is that, given T, the algorithm infers a suitable dictionary and encodes T accordingly. The problem with compressed indexes based on this approach is that the encoding of T is not suitable for pattern matching. In fact the dictionary generated by the Ziv-Lempel algorithm is dynamically updated at the same time that T is processed. This means that the same string may be encoded in several different ways, since the dictionary changes from one occurrence, of the string, to another. This results in an undesirable encoding. The solution to this problem forces us to destroy the on-line property of the Ziv-Lempel algorithm. Our algorithm runs in two phases: in the first one we use the LZ78 algorithm to infer a dictionary; in the second one we organize T in an off-line way, using the inferred dictionary.

We start our exposition with some basic concepts and a general description of our index, based on generic dictionaries. Afterwards, we show how to use the information from the LZ78 algorithm to produce a suitable dictionary and prove that we obtain a compressed full-text self-index. Next we describe some of the practical decisions that were taken to implement our algorithm. Finally, we show some experimental results and conclusions.

2 Basic Concepts and Notation

For basic concepts related to strings and suffix trees we refer the reader to one of the many good references available, e.g. Gusfield [11]. We use the following conventions: strings are sequences of letters from the alphabet Σ, of size σ, and start at index position 0; prefixes, substrings and suffixes are denoted respectively as $S[..i]$, $S[i..j]$, $S[j..]$;a set C is suffix/prefix if any suffix/prefix of an element of C is also an element of C; m is the size of the pattern string P, u is the size of the text string T and occ is the number of occurrences of P in T. By suffix tree we refer to a generalized suffix tree. The terminator symbols are not considered as part of the edge-labels. The suffix trie is the uncompressed version of the suffix tree, i.e. it contains a node between any two letters in a label. A point is a node in the suffix trie. We refer indifferently to points in a suffix tree and to their path-labels. $\textsc{Sdep}(p)$ is the string depth of point p. $\textsc{Father}(v)$ is the father node of node v. $\textsc{SuffixLink}(v)$ is the node pointed by the suffix link of node v. $\textsc{Letter}(v, i)$ equals $v[i]$, i.e. the i-th letter of the path-label of node v. $\textsc{Descend?}(p, c)$ is true iff it is possible to descend from point p with c and $\textsc{Descend}(p, c)$ returns the resulting point. In a suffix tree the first letters of every edge are referred to as **branching letters**. By $\textsc{Dfs}(v)$ we refer to the

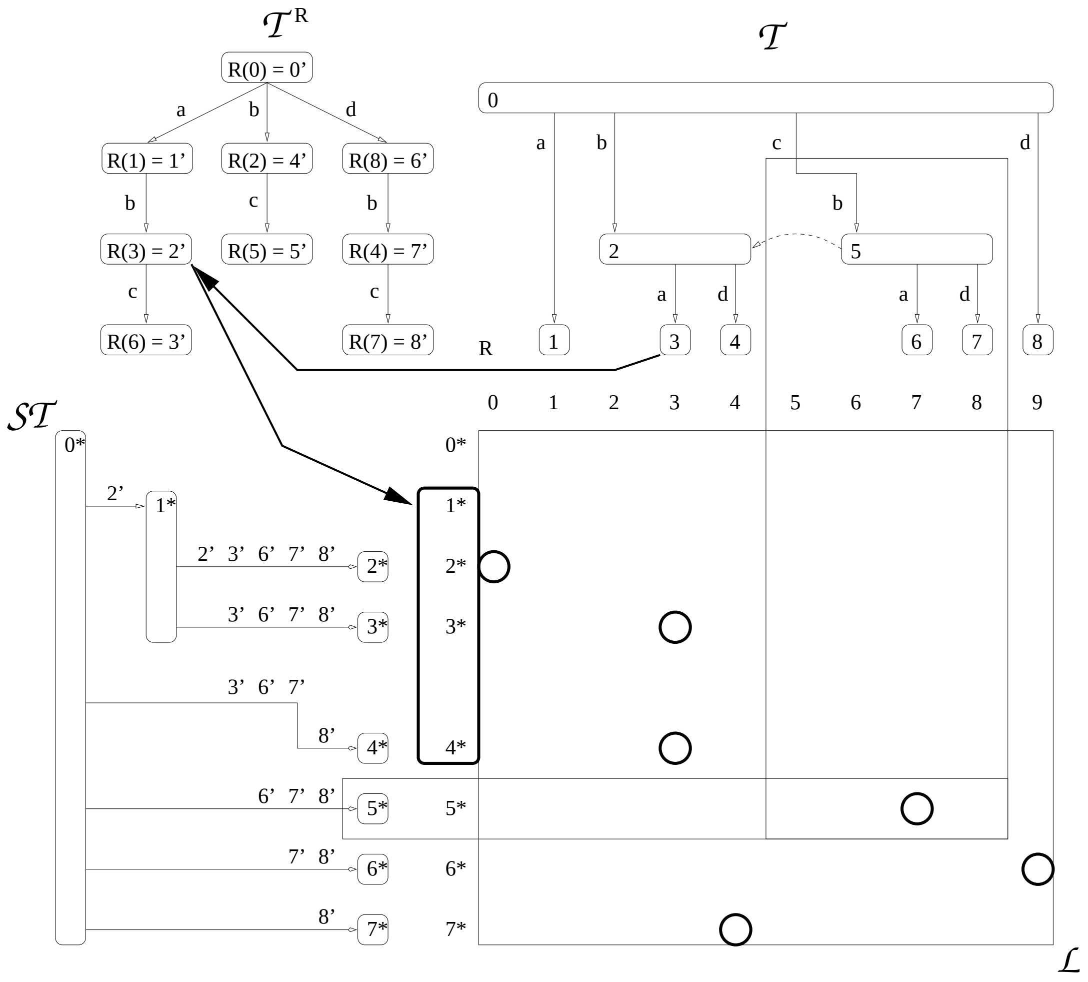

Fig. 1 (top-right) Suffix tree for strings $\{a, b, ba, bd, cba, cbd, d\}$. Suffix link from cb to b shown by a dashed arrow. Nodes show their DFS value in $\mathcal{T}$. (top-left) Reverse tree of the suffix tree on the right. Nodes show their DFS value in $\mathcal{T}^R$. The R mapping is shown and $R(3)$ is indicated by a bold arrow. (bottom-left) Sparse suffix tree of T, nodes show their $\text{DFS}_{\mathcal{ST}}$ values. Weak descent $W(\text{ROOT}_{\mathcal{ST}}, 2')$ shown in bold rectangle. (bottom-right) Linking points over spaces supported by DFS’ and $\text{DFS}_{\mathcal{ST}}$ values. Orthogonal range query [5*,5*]:[5,8].

depth-first time-stamp [4] of a node v in a suffix tree and by DFS’(p) to the depth-first time-stamp of a point p in a suffix trie.

Definition 1 *The* ***range*** $I(p)$ *of a point* p *of a suffix tree* $\mathcal{T}$ *is the interval of the* DFS’ *values of the points that are descendants of* p.

As a running example consider $T = cbdbddcbababa$ and $\mathcal{T}$ as the suffix tree in Figure 1 (top-right). In our example $\text{DFS}(c)$ is undefined, $\text{DFS}(cb) = 5$, $\text{DFS'}(c) = 5$, $\text{DFS'}(cb) = 6$ and $I(c) = [5, 8]$. Table 1 presents the main symbols used throughout this paper.

2.1 Descend and Suffix Walk

Descend and suffix walks are classical algorithms over suffix trees but since they constitute an important component of our method we will briefly explain them here.

An element that is responsible for the flexibility of suffix trees is the suffix link. The **suffix-link** of a node v of a suffix tree is a pointer to node $v[1..]$, denoted by $\text{SUFFIXLINK}(v)$. We define in an

Table 1 Main symbols used

Symbol	Meaning
T	Text string
u	Original length of text string in characters, i.e. $\|T\|$
P	Pattern string
m	Length of pattern string in characters, i.e. $\|P\|$
Σ	Alphabet for P and T
σ	Alphabet size, $\sigma = \|\Sigma\|$
occ, occ_1, $occ_{>1}$	Number of occurrences of the pattern in the text, inside a block and spanning more than one block respectively
occ'	occurrences determined by an orthogonal range query
H_k	k-th order entropy of a text character
i, j	counters in the Descend and Suffix Walk algorithm or generic indexes
Z_i	Ziv-Lempel block
n	Number of LZ78 blocks of the text
ϵ	either the empty string or a small positive real number
$\mathcal{T}_{78}$	Ziv-Lempel suffix tree, dictionary
d, t	number of nodes/points in the Ziv-Lempel suffix tree,
t	the tree depth in the FOR variant
$\mathcal{ST}_{78}$	Ziv-Lempel sparse suffix tree
d'	number of nodes in the Ziv-Lempel sparse suffix tree
$\mathcal{T}_{78}(T)$	$\mathcal{T}_{78}$-maximal parsing
f	size of the $\mathcal{T}_{78}$-maximal parsing
R	reverse mapping between trees
V	Descend and Suffix walk variant and block bitmap

artificial way SUFFIX_LINK(ROOT) as a node that descends to the root by every letter including terminator symbols. Several suffix tree algorithms use suffix links. One such algorithm is a greedy traversal of the tree, greedy in the sense that the algorithm traverses the tree trying to maximize the string depth at all times. Suppose we are given pattern P and a suffix tree $\mathcal{T}$. A greedy traversal of P in $\mathcal{T}$ consists in trying to read a string P by starting from the root and descending as much as possible. When it is impossible to descend any further, we follow suffix-links until descending becomes possible again.

Definition 2 *The* ***descend and suffix walk*** *of a string P over a suffix tree $\mathcal{T}$ is the sequence $p_0 \ldots p_{2m}$ of points of $\mathcal{T}$ computed by Algorithm 1, i.e. the sequence of values taken by the variable* point*.*

It is important to notice that Algorithm 1 starts by appending to P a new terminator character $\$'$ that fails to match with any other character. The following lemma explains why, for each value of i, the point values at line 5 correspond to the largest suffix of $P[..i-1]$ that is a point in $\mathcal{T}$.

Lemma 1 (For Invariant) *Before any execution of line 5 of Algorithm 1, it is always true that for any $j' < j$ we have that $P[j'..i-1]$ is not a point in $\mathcal{T}$.*

Proof First it should be obvious that, except in line 10, $point = P[j..i-1]$, since the SUFFIXLINK (resp. DESCEND) and j++ (resp. i++) instructions are consecutive.

The lemma is proved by induction on i. The base is trivial. We assume that before line 7 is executed if $j' < j$ then $P[j'..i]$ is not a point in $\mathcal{T}$. Our result follows immediately from this property by observing that the *point* and i are updated before reaching line 5 again.

The previous property can be proved by induction on the number of times the while loop ran on an iteration of the for loop. The base follows from the induction hypotheses of the lemma, by observing that, since $\mathcal{T}$ is suffix closed, if point $P[j'..i-1]$ is not in $\mathcal{T}$, neither is point $P[j'..i]$. Finally assume that the while's guard is true, i.e. NOT DESCEND?($P[j..i-1]$, $P[i]$). Therefore $P[j..i]$ is not a point in $\mathcal{T}$. Hence if $j' < j+1$ then $P[j'..i]$ is also not a point in $\mathcal{T}$. □

Algorithm 1 Descend and Suffix Walk Algorithm

```
 1: procedure DESCEND&SUFFIX(P)
 2:     P ← P.$'
 3:     j ← 0
 4:     point ← ROOT
 5:     for i ← 0, i < |P|, i++ do
 6:         trace_left[i] ← point
 7:         while NOT DESCEND?(point, P[i]) do
 8:             trace_right[j] ← point
 9:             j++
10:             point ← SUFFIXLINK(point)
11:         end while
12:         point ← DESCEND(point, P[i])
13:     end for
14: end procedure
```

i	0	1	2	3	4	5	6	7
P[i]	c	b	d	b	d	d	c	$'
trace_left[i]	ϵ	c	cb	cbd	b	bd	d	c
DFS'(father_left[i])	0	0	6	8	2	4	9	0
DFS'(trace_left[i])	0	5	6	8	2	4	9	5
DFS'(child_left[i])	0	6	6	8	2	4	9	6
trace_right[i]	cbd	bd	d	bd	d	d	c	ϵ
DFS'(father_right[i])	8	4	9	4	9	9	0	0
DFS'(trace_right[i])	8	4	9	4	9	9	5	0
I(trace_right[i])	[8,8]	[4,4]	[9,9]	[4,4]	[9,9]	[9,9]	[5,8]	[0,9]
DFS'(child_right[i])	8	4	9	4	9	9	6	0
P[i..]	cbd.bd.d.c	bd.bd.d.c	d.bd.d.c	bd.d.c	d.d.c	d.c	c	ϵ
tail(P[i..])	c	c	c	c	c	c	c	ϵ
H(P[i..])	748	448	848	48	88	8	ϵ	ϵ
R(H(P[i..]))	6'7'8'	udef	udef	6'7'	6'6'	6'	ϵ	ϵ
|father_left[i]| == i		FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
W(R(H(P[i..])), R(father_left[i]))			$\emptyset$	[5*,5*]	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
I(tail(P[i..]))		[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[0,9]
occ'			0	1	0	0	0	0

Table 3 (Top) Descend and suffix walk of *cbdbddc* in $\mathcal{T}$. (Bottom) Values for locating type > 1 occurrences.

This lemma shows that the value of the point in line 6 is left maximal, i.e. no $P[j'..i-1]$ with $j' < j$ is a point of the suffix tree. Likewise the points in line 8 are right maximal, since the while's guard has just evaluated true. This gives a way to classify the points that were reached by the descend and suffix walk.

Definition 3 *The* ***left and right traces*** *of a string P over a suffix tree $\mathcal{T}$ are the sub-sequences of the descend and suffix walk given respectively by lines 6 and 8 of Algorithm 1.*

By **father_right**$[i]$ (resp. **father_left**$[i]$), we refer to the lowest ancestor of $trace\_right[i]$ (resp. $trace\_left[i]$) that is a node of $\mathcal{T}$ and by **child_right**$[i]$ (resp. **child_left**$[i]$), to the highest descendant of $trace\_right[i]$ (resp. $trace\_left[i]$) that is a node of $\mathcal{T}$. Table 3 (top) shows the descend and suffix walk of *cbdbddc* in $\mathcal{T}$.

We will now explain why Algorithm 1 runs in $O(m)$ time. First it should be clear that Algorithm 1 does terminate.

```
              1
    012345678901234567
B:  110110100110100100
    (()(()())(()())())
```

Fig. 2 Sample bitmap that represents the suffix tree $\mathcal{T}$ in Figure 1 (top-right)

Theorem 1 *Expression $V(i) = 3m - i - 2j - t$ is a variant of the for loop, where t is the tree depth of the point. Therefore Algorithm 1 terminates.*

Proof Suppose that $V(i) \leq 0$. Since $t \leq i - j$, then $3m - 2i - j \leq 3m - i - 2j - t$. Since $j \leq i$, then $3m - 3i \leq 3m - 2i - j$. Therefore $3m - 3i \leq 0$, hence $m \leq i$ and the for cycle terminates.

Except for instruction 10, it should be evident that $\Delta V = V(i+1) - V(i) < 0$ for any i, since j is non-decreasing and i is strictly increasing for each iteration of the FOR loop. The problem with the SUFFIX_LINK operation is that it may cause t to decrease. However t can decrease at most by 1. The factor 2 associated with j compensates this effect. Therefore in every iteration of the while cycle $\Delta V < 0$. □

For now we assume that the operations DESCEND and DESCEND? are computed in constant time and later give a more realistic analysis. The problem of analyzing the time of Algorithm 1 is that operation SUFFIXLINK is computed for points, not just nodes, and therefore does not necessarily run in constant time.

Lemma 2 (Skip/count trick) *The* SUFFIX_LINK *function runs in $O(\Delta t + 2)$ time, where Δt is the variation of tree depth.*

Proof Computing the SUFFIX_LINK for the nodes of $\mathcal{T}$ can be done in $O(1)$ by storing the suffix-links in $\mathcal{T}$. For a point, the idea is to first use the suffix link of its father node and then descend until the string depth is equal to the string depth of the original point less 1. In order to descend, it is not necessary to read the complete edge labels. The reason is that $P[j+1..i-1]$ must be a point in $\mathcal{T}$ since $P[j..i-1]$ is. Therefore we only need to check the branching letters of the nodes we find along the way. Hence, we conclude that this procedure can be computed in $O(\Delta t + 2)$ time. □

Observe that $-\Delta V$ counts all the operations executed in an iteration of the for loop, including the time to compute SUFFIX_LINK. Therefore Algorithm 1 runs in $O(V(0)) = O(m)$ time.

3 Succinct Data structures

By bitmap B we refer to a string over $\{0, 1\}$ of length $|B|$. Fundamental tools to produce succinct data structures are the RANK and SELECT operations over bitmaps. The operation RANK(B, i) counts the number of 1's in $B[..i-1]$ and SELECT(B, i) returns the smallest j such that RANK$(B, j+1) = i$, i.e. the position of the i-th 1. For the example bitmap in Figure 2, we have that RANK$(B, 3) = 2$ and SELECT$(B, 2) = 1$. Munro [17] and Clark showed how to support these operations in $O(1)$ time and $|B| + o(|B|)$ bits. Succinct data structures can also be combined with data compression techniques when B is compressible, solutions that require $|B|H_0(B) + o(|B|)$ bits may be more adequate. This line of work was initiated by Pagh [23] and extended by Raman et al. [24].

3.1 Succinct Suffix Trees

Since our approach is based on suffix trees, we need an adequate succinct representation for them. We have already mentioned that trees can be represented as a sequence of parentheses, i.e. can be represented as a bitmap. For example the bitmap in Figure 2 represents the suffix tree $\mathcal{T}$ in Figure 1 (top-right). For example, the node with DFS value 2 is represented by the parentheses at positions

3 and 8 of B. The DFS value can be obtained from B as $\textsc{Rank}(B, 3) + B[3] - 1 = 2 + 1 - 1 = 2$. This is the computation performed by the LEFTRANK operation, i.e. LEFTRANK corresponds to DFS. The $\textsc{RightRank}(v)$ corresponds to the largest DFS value among the descendants of v. This operation can be computed as $\textsc{Rank}(B, 8) + B[8] - 1 = 5 + 0 - 1 = 4$. This is consistent with Figure 1, where the node with DFS value 4 is the last descendant of the node with DFS value 2.

The RANK and SELECT operations also proved useful for representing permutations. Munro et al. [18] showed how to represent a permutation of d elements and its inverse in $(1+\epsilon)d\log d + o(d)$ bits, where ϵ is constant and $0 < \epsilon \leq 1$. An element of the permutation can be computed in $O(1)$ and an element of the inverse in $O(1/\epsilon)$.

Geary et al. [7] presented a succinct representation of ordinal d-node trees in $2d + o(d)$ bits, supporting, among others, the following operations in constant time:

- $\textsc{Anc}(v, j)$ returns the j-th ancestor of node v (for example $\textsc{Anc}(v, 1)$ is $\textsc{Father}(v)$);
- $\textsc{LeftRank}(v)$ returns $\textsc{Dfs}(v)$;
- $\textsc{RightRank}(v)$ returns the largest DFS value among the descendants of v;
- $\textsc{Select}(j)$ returns the node with DFS time j;
- $\textsc{Child}(v, j)$ returns the j-th child of node v;
- $\textsc{Deg}(v)$ returns the number of children of node v;
- $\textsc{Depth}(v)$ returns the tree depth of node v.

Definition 4 *The* ***reverse tree*** $\mathcal{T}^R$ *of a suffix tree* $\mathcal{T}$ *is the minimal labeled tree that, for every node* v *of* $\mathcal{T}$*, contains a node* v^R*, where* v^R *denotes the reverse string of* v*.*

The tree $\mathcal{T}^R$ is shown in Figure 1 (top-left). Observe for example that, since *cbd* is a node of $\mathcal{T}$, there is a node $cbd^R = dbc$ in $\mathcal{T}^R$. We define a canonical mapping R that, for every node v in $\mathcal{T}$, maps $\textsc{Dfs}(v)$ to $\textsc{Dfs}(v^R)$ (see Figure 1). We will use $R(v)$ to denote $R(\textsc{Dfs}(v))$. Note that since the nodes of $\mathcal{T}$ form a suffix closed set, the nodes of $\mathcal{T}^R$ form a prefix closed set.

We assume that the tree structure of $\mathcal{T}$ and $\mathcal{T}^R$ are stored using the previous representation. Arroyuelo et al. [1] proposed a way to represent the R mapping. Since R is a permutation, R and R^{-1} can be stored using the representation of Munro et al. [18] in $(1+\epsilon)d\log d + o(d)$ bits, where ϵ is fixed and $0 < \epsilon \leq 1$. This way R and R^{-1} can be computed in $O(1)$ and $O(1/\epsilon)$ time respectively.

Lemma 3 *A suffix tree* $\mathcal{T}$ *with* d *nodes can be stored in* $(1+\epsilon)d(\log d) + 5d + o(d)$ *bits. Let* p *be a point,* c *a letter and* v *a node of* $\mathcal{T}$*. This representation provides the operations given by Geary et al. in* $O(1)$ *time. Moreover it provides* $\textsc{Sdep}(v)$ *in* $O(1)$ *time,* $\textsc{Suffix\_Link}(v)$*,* $\textsc{Letter}(v, i)$*, in* $O(1/\epsilon)$ *time and* $\textsc{Descend?}(p, c)$*,* $\textsc{Descend}(p, c)$ *in* $O((\log\sigma)/\epsilon)$ *time.*

Proof According to our notation $R(v)$ represents $\textsc{Select}_{\mathcal{T}^R}(R(\textsc{LeftRank}(v)))$. Observe that $\textsc{Sdep}(v)$ can be computed as $\textsc{Depth}_{\mathcal{T}^R}(\textsc{Select}_{\mathcal{T}^R}(R(\textsc{LeftRank}(v))))$ which can be represented as $\textsc{Depth}_{\mathcal{T}^R}(R(v))$, since $\mathcal{T}^R$ is prefix closed. The operation $\textsc{Suffix\_Link}(v)$ is computed as $R^{-1}(\textsc{Father}_{\mathcal{T}^R}(R(v)))$. Observe that $v[0]$ represents the letter just below the root. For example $cbd[0] = c$. We define a bitmap D to compute $v[0]$, in a way similar to Sadakane [27]. We have that $D[0] = 1$ and, for $i > 0$, $D[i] = 0$ iff $\textsc{Dfs}(v) = i$, $\textsc{Dfs}(v') = i + 1$ and $v[0] = v'[0]$. In our example $D = 11001001$. We can compute $v[0]$, when v is not the ROOT, in $O(1)$ as the letter in position $\textsc{Rank}_1(D, \textsc{Dfs}(v))$ of Σ. This requires $d + o(d)$ bits. The operation $\textsc{Letter}(v, i)$ can be computed from $R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v), i))$. This expression represents following enough suffix links to make the letter we want appear just below the root, i.e. $\textsc{Letter}(v, i) = R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v), i)[0]$. When p is not a node, $\textsc{Descend?}(p, c)$ can be computed in $O(1/\epsilon)$ time by consulting LETTER for the point below p. If p is a node, we do a binary search among the children of p. If we find a child that starts with c, we return true. Procedure $\textsc{Descend}(p, c)$ updates the value of p. When p is a point, this is done in $O(1)$ time. When p is a node, we first proceed as in DESCEND?. □

Finally observe that with this representation we cannot compute $\textsc{Dfs'}(v)$. The DFS' values are essential to our algorithm because they serve as a supporting space for range queries. This result can be obtained with a compressed bitmap.

Lemma 4 *For a suffix tree $\mathcal{T}$ with t points and $2n$ nodes, operations* $\textsc{Dfs}'(p)$ *and* $I(p)$ *can be computed in $O(1)$ time using $tH_0 + O(t \log\log t/\log t)$ extra bits, where H_0 is the empirical entropy of a bitmap with $(t-2n)$ ones and $2n$ zeros.*

Proof Consider the bitmap that for every point of $\mathcal{T}$ stores 1 if the corresponding point is a node and 0 if it is not a node. The bitmap is sorted in DFS' order. Using the compressed representation of Raman et al. [24] this bitmap can be stored in $tH_0 + O(t \log\log t/\log t)$ bits supporting $\textsc{Select}_1$ in $O(1)$ time. Observe that for a node v we have that $\textsc{Dfs}'(v) = \textsc{Select}_1(\textsc{Dfs}(v))$.

For a point p, $\textsc{Dfs}'(p)$ is computed as $\textsc{Dfs}'(v) - \textsc{Sdep}(v) + \textsc{Sdep}(p)$, where v is the highest node that is a descendant of p[1]. Also $I(p) = [\textsc{Dfs}'(p), \textsc{Dfs}'(\textsc{Select}(\textsc{RightRank}(v)))]$. □

3.2 Wavelet Trees

Wavelet trees are a recurrent succinct data structure. They were proposed by Grossi et al. [9] as a structure for supporting RANK and SELECT for sequences over an alphabet larger than 2. They were also proposed by Chazelle [3] for performing orthogonal range queries. Obviously the algorithms over the structure are different. However, both use RANK and SELECT over bitmaps. This description of the structure given by Chazelle was pointed out by Mäkinen et al. [20].

Consider for example the sequence $0, 3, 3, 7, 9, 4$. The wavelet tree of this sequence is shown in Figure 3. The wavelet tree is a perfect binary tree of height $\lceil \log \sigma \rceil$. Each node stores a sub-sequence of the original sequence. The root stores the whole sequence. Starting from the most significant bit, the left node stores the sub-sequence for which this bit is 0, the right node stores the sub-sequence for which this bit is 1. In our example the left sub-sequence is $0, 3, 3, 7, 4$ and the right sub-sequence is 9. This process continues until all bits have been used. To descend from one node to a child node we use the RANK operation. For the left node we use $\textsc{Rank}_0$ and for the right node $\textsc{Rank}_1$. In our example we can track the element 4 by computing $\textsc{Rank}_0(5) = 4$ at the root node. Note that element 4 is in position 4 of the left child of the root. Obviously moving upwards uses the inverse procedure, i.e. the $\textsc{Select}_0$ operation. Every leaf of the wavelet tree represents a type of element in the sequence. Moving from the root to a leaf allows us to compute rank for the element associated with the leaf. Conversely, moving from a leaf to the root allows us to compute SELECT for that element.

The tree structure is only conceptual. In fact the only information that is stored are the bitmaps highlighted in Figure 3. Further RANK and SELECT operations can be used to delimit the bits that correspond to a given node of the tree.

The wavelet tree can also be used to compute orthogonal range queries. Consider a grid $[1, f] \times [1, f]$ with f points inside. An orthogonal range query consists in determining the points inside a rectangle (see Figure 1). Provided that the points are all distinct in the first coordinate they can stored in a wavelet tree, by building a list of the second coordinate values ordered by the first coordinate. In the example of Figure 1 the resulting sequence is $0, 3, 3, 7, 9, 4$. This requires $f \log f(1 + o(1))$ bits. In fact it is easy to extend the space of the second coordinate, i.e. extend the space to $[1, f] \times [1, f']$. This will require $f \log f'(1 + o(1))$ bits instead. To compute a range query $[i, i'] \times [j, j']$ we start by locating the range $[i, i']$ at the root of the wavelet tree. When we descend we track the elements i and i'. The idea is to track every path that is contained in the $[j, j']$ range. Obviously we can avoid descending by nodes for which the corresponding range $[i, i']$ is empty. Therefore whenever a leaf is reached an occurrence is found, i.e. it takes $O((1 + occ') \log f')$ time to report occ' occurrences. A simpler procedure can be used to count the number of occurrences in range $[i, i'] \times [j, j']$. The procedure consists in descending by j and j', the total of occurrences associated with the non-shared part of these paths gives the number of occurrences. This takes $O(\log f')$ time.

The ranges we are going to use are obtained from other structures in our index, and, in particular, from suffix trees.

[1] Note that we assume that v is part of the representation of p.

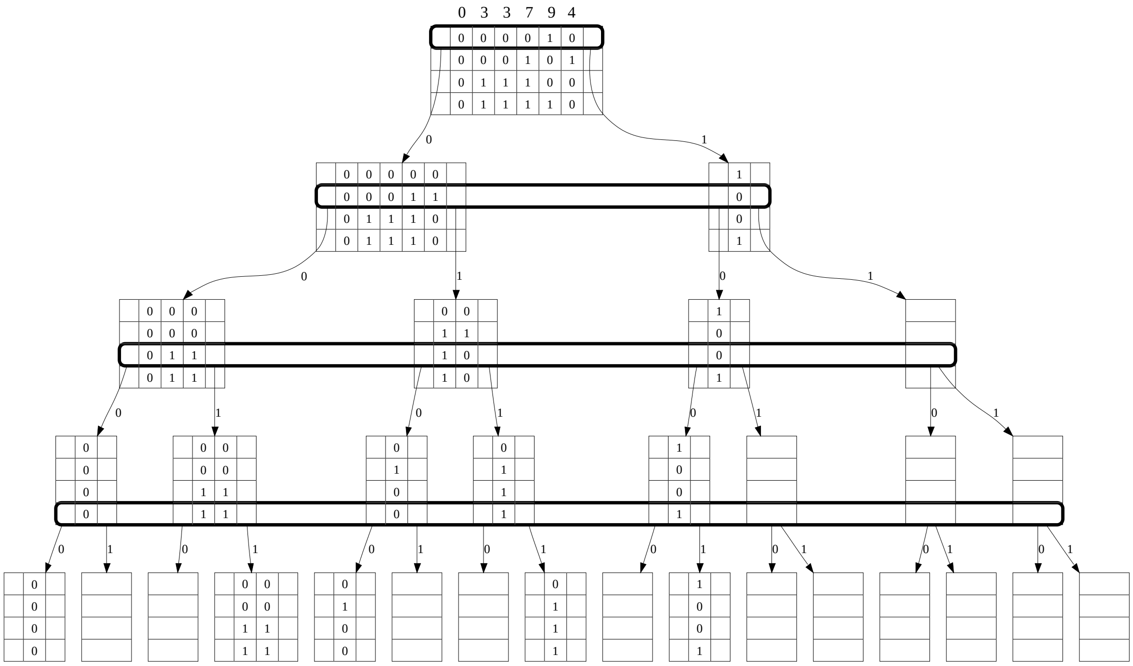

Fig. 3 Wavelet tree for sequence $0, 3, 3, 7, 9, 4$.

4 A Full-Text Index Using Suffix Tree Dictionaries

In this section we explain the main contribution of this paper. Our data structure is very similar to an inverted file. We will use this similarity to provide insight into the algorithm.

4.1 Generic Inverted Index

Throughout section 3 we assume that we are given an arbitrary suffix tree $\mathcal{T}$ with d nodes, that we will use as a dictionary. We consider as dictionary *words* the path-labels of the nodes of $\mathcal{T}$. The first thing we should do is to organize T according to our dictionary $\mathcal{T}$, much like what is done in inverted files when given a lexicon.

Definition 5 *The* ***$\mathcal{T}$-maximal parsing*** *of string T is the sequence of nodes $v_1, \ldots, v_f$, whose concatenated path-labels compose T, i.e. $T = v_1 \ldots v_f$, and for every j, v_j is the largest prefix of $v_j \ldots v_f$ that is a node of $\mathcal{T}$.*

We assume that $\mathcal{T}$ is appropriate for T, i.e. that it is possible to parse T in a maximal way. In our example, the $\mathcal{T}$-maximal parsing of a string T is the sequence cbd, bd, d, cba, ba, ba. We refer to the elements of the $\mathcal{T}$-maximal parsing of T as *blocks*. Note that the strings in the dictionary appear in the $\mathcal{T}$-maximal parsing. We denominate them as words when referring to the dictionary and as blocks when referring to the $\mathcal{T}$-maximal parsing. We will store the $\mathcal{T}$-maximal parsing of T in compact form as a string of numbered blocks.

Definition 6 *The* ***translation*** *$V(v_1, \ldots, v_f)$ of a sequence $v_1, \ldots, v_f$ of nodes is the string* $\textsc{Dfs}(v_1) \ldots \textsc{Dfs}(v_f)$.

We denote by $\mathcal{T}(T)$ the translation of the $\mathcal{T}$-maximal parsing of T. Since the $\mathcal{T}$-maximal parsing of T is the sequence cbd, bd, d, cba, ba, ba, its translation is the string $\mathcal{T}(T) = 748633$. Note that word ba is associated with two blocks, v_5 and v_6.

Inverted files usually store a list of occurrences for every word of the dictionary. To play this role we will use a stronger indexing structure, a sparse suffix tree. For reasons that will become clear in Section 5 we must reverse the string $\mathcal{T}(T)$. This is achieved by extending the canonical mapping R to sequences in the following way: $R(v_1 \dots v_f) = R(v_f) \dots R(v_1)$. In our example $R(\mathcal{T}(T)) = R(748633) = R(3)R(3)R(6)R(8)R(4)R(7) = 2'2'3'6'7'8'$. This corresponds to the notion of reverse string, because the concatenation of the path-labels of $R(\mathcal{T}(T))$ in $\mathcal{T}^R$ is $ab.ab.abc.d.db.dbc = T^R$.

Definition 7 *The* ***sparse suffix tree***[2] $\mathcal{ST}$ *of a string* T *and a suffix tree* $\mathcal{T}$ *is the suffix tree of* $R(\mathcal{T}(T))$.

The sparse suffix tree of our example is shown in Figure 1 (bottom-left). We can descend in the sparse suffix tree in the usual way with $\textsc{Descend}_{\mathcal{ST}}$. However, since $\mathcal{T}^{\mathcal{R}}$ provides the alphabet for $\mathcal{ST}$, we can also take that into consideration when descending.

Definition 8 *The* ***weak descent*** $W(p, v^R)$ *for a point* p *in* $\mathcal{ST}$ *and a node* v^R *in* $\mathcal{T}^R$ *is the interval of* $\textsc{Dfs}_{\mathcal{ST}}$ *values of the nodes below the following points:*
$\{p.\textsc{Dfs}_{\mathcal{T}^R}(v') \mid v' \text{ is a descendant of } v^R \text{ in } \mathcal{T}^R\}$

Note that by $p.\textsc{Dfs}_{\mathcal{T}^R}(v')$ we are referring to the points whose path labels result from concatenating the letter $\textsc{Dfs}_{\mathcal{T}^R}(v')$ with the path-label of point p.

For example, $W(\textsc{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, since this contains the $\textsc{Dfs}_{\mathcal{ST}}$ values for the nodes below $2', 3'$ in $\mathcal{ST}$ (see Figure 1). This can be computed in $O((\log d)/\epsilon)$ time. We perform two binary searches in the children of p, searching for $\textsc{LeftRank}_{\mathcal{T}^R}(v)$ and $\textsc{RightRank}_{\mathcal{T}^R}(v)$. Then $W(p, v^R) = [\textsc{LeftRank}_{\mathcal{ST}}(v''), \textsc{RightRank}_{\mathcal{ST}}(v''')]$, where v'' and v''' are the nodes found by the binary searches.

In order to find occurrences of strings across more than one block, we will need to store the relations across contiguous blocks. This motivates the following two definitions.

Definition 9 *The* ***head, tail*** *of the* $\mathcal{T}$*-maximal parsing are respectively sequence* $v_1, \dots, v_i$ *and string* $v_{i+1} \dots v_f$ *such that* $v_1, \dots, v_i$ *is the smallest sequence for which* $v_{i+1} \dots v_f$ *is a point in* $\mathcal{T}$.

We denote by $H(T)$ the translation of the head of the $\mathcal{T}$-maximal parsing of T. The head of the $\mathcal{T}$-maximal parsing of T is cbd, bd, d, cba, ba and the tail is the string ba. Hence $H(T)$ equals 74863. It may seem that **tail** is always just v_f. Consider a modification $\mathcal{TM}$ of tree $\mathcal{T}$ were node cbd is replaced by $cbde$ and nodes bde, de, e are added to complete the suffix tree. Note that cbd is not a node of $\mathcal{TM}$, as it is only a point. The string, $bcbd$ is parsed as $b.cb.d$ and the tail is $cb.d$ and, therefore, it is not just the last block.

Next we define a set of points relating the leaves of $\mathcal{ST}$ with the points in $\mathcal{T}$.

Definition 10 *The* ***linking points set*** *of the* $\mathcal{T}$*-maximal parsing* $v_1 \dots v_f$ *of* T *is the following set:*

$$\mathcal{L} = \left\{ \langle \textsc{Dfs}(R(V(v_1 \dots v_i))), \textsc{Dfs'}(p_i) \rangle \;\middle|\; \begin{array}{l} p_i \text{ is the largest prefix of } v_{i+1} \dots v_f \\ \text{that is a point in } \mathcal{T}, \text{ for } 0 < i \leq f \end{array} \right\}$$

The set $\mathcal{L}$ is shown in Figure 1 (bottom-right) and consists of the following points:

- $\langle \textsc{Dfs}(R(V(cbd, bd, d, cba, ba, ba))), \textsc{Dfs'}(\epsilon) \rangle = \langle \textsc{Dfs}(2'2'3'6'7'8'), 0 \rangle = \langle 2^*, 0 \rangle$
- $\langle \textsc{Dfs}(R(V(cbd, bd, d, cba, ba))), \textsc{Dfs'}(ba) \rangle = \langle \textsc{Dfs}(2'3'6'7'8'), 3 \rangle = \langle 3^*, 3 \rangle$
- $\langle \textsc{Dfs}(R(V(cbd, bd, d, cba))), \textsc{Dfs'}(ba) \rangle = \langle \textsc{Dfs}(3'6'7'8'), 3 \rangle = \langle 4^*, 3 \rangle$
- $\langle \textsc{Dfs}(R(V(cbd, bd, d))), \textsc{Dfs'}(cba) \rangle = \langle \textsc{Dfs}(6'7'8'), 7 \rangle = \langle 5^*, 7 \rangle$
- $\langle \textsc{Dfs}(R(V(cbd, bd))), \textsc{Dfs'}(d) \rangle = \langle \textsc{Dfs}(7'8'), 9 \rangle = \langle 6^*, 9 \rangle$
- $\langle \textsc{Dfs}(R(V(cbd))), \textsc{Dfs'}(bd) \rangle = \langle \textsc{Dfs}(8'), 4 \rangle = \langle 7^*, 4 \rangle$

To compute orthogonal range queries we use the wavelet tree as described. As referred, this structure requires $f \log f'(1 + o(1))$ bits and can compute orthogonal range queries in the space $[1, f] \times [1, f']$ in $O((1 + occ') \log f')$ time. We need to store points in the $[0, d' - 1] \times [0, t - 1]$ space, where d' is the number of nodes of $\mathcal{ST}$. We only need to store f points. Therefore we must reduce the support space to the rank space. The space $[0, d' - 1]$ can be reduced to $[1, f]$ in $O(1)$ time,

[2] Similar to a concept defined by Kärkkäinen et al. [14].

with RANK over a bitmap of $d' + o(d')$ bits. The second reduction is obtained by setting f' to t and, therefore, time to report occurrences is $O((1 + occ') \log t)$.

We propose an index data structure composed of the dictionary $\mathcal{T}$, the sparse suffix tree $\mathcal{ST}$ and the linking points $\mathcal{L}$. We will now explain how to use this index to solve the exact matching problem. Our search algorithm proceeds differently depending on whether the pattern is completely contained inside a block or spans more than one block. We refer to this as **type** 1 and **type** > 1 occurrences.

4.2 Occurrences Lying Inside a Single Block

The algorithm for finding occurrences inside a single block starts by identifying all the words in the dictionary $\mathcal{T}$ that contain P as a substring. Since $\mathcal{T}$ is a suffix tree, it is possible to achieve this in a simple way.

– Descend by P in $\mathcal{T}$. If this is impossible then there are no type 1 occurrences of P.
– Start a depth-first traversal of the sub-tree below P.
– For each node v reached compute the range query $W(\textsc{Root}_{\mathcal{ST}}, R(v)) : [0, t]$.

The search in $\mathcal{T}$ consists in considering words that start with P and appending some letters. The weak descend and the range query consist in prepending some letters to the words found on the search in $\mathcal{T}$. For example, consider $P = b$. By reading b, we reach node 2 of $\mathcal{T}$ (see Figure 1). The search on $\mathcal{T}$ returns nodes $2, 3, 4$, hence it leads us to consider words b, ba, bd. This originates the following weak descends: $W(\textsc{Root}_{\mathcal{ST}}, 4') = \emptyset$, $W(\textsc{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, $W(\textsc{Root}_{\mathcal{ST}}, 7') = [6^*, 7^*]$. We do not need to consider words that start with b, since they do not correspond to blocks; there may be occurrences of *ba* or *cba* because of *ba*; there may be occurrences of *bd* and *cbd* because of *bd*. The range queries return no occurrences for b, occurrences 2^*, 3^* and 4^* for *ba* and occurrences 6^* and 7^* for *bd*. This corresponds to occurrences *cbd.bd.d.cba.ba.<u>b</u>a*, *cbd.bd.d.cba.<u>b</u>a.ba*, *cbd.bd.d.c<u>b</u>a.ba.ba* for *ba* and occurrences *cbd.<u>b</u>d.d.cba.ba.ba*, *c<u>b</u>d.bd.d.cba.ba.ba*, for *bd*.

Theorem 2 *The above procedure is correct and complete.*

Proof (Correct) Clearly every reported block is $\alpha.P.\beta$ for some α,β and hence it contains an occurrence of P. (Complete) Suppose block $v_i = \alpha.P.\beta$. Hence $\alpha.P.\beta$ is a node in $\mathcal{T}$. Since $\mathcal{T}$ is a suffix tree, $P.\beta$ is also a node in $\mathcal{T}$. Node $P.\beta$ is reached by the search in $\mathcal{T}$, since it starts by P. Every node v of $\mathcal{ST}$ for which $v[0] = \textsc{Dfs}((\alpha.P.\beta)^R)$ has its $\textsc{Dfs}_{\mathcal{ST}}$ time in $W(\textsc{Root}_{\mathcal{ST}}, (P.\beta)^R)$. Hence block v_i is found in the range query. □

This algorithm was essentially presented by Navarro [21], except for the fact that the range queries were computed as depth-first searches in a trie similar to $\mathcal{T}^R$. In Navarro's algorithm, each node of that trie stored one block. Therefore the time of these searches was bounded by the number of type 1 occurrences of P, denoted by occ_1. We do not have a direct correspondence between the nodes of $\mathcal{T}^R$ and the blocks of $\mathcal{T}$-maximal parsing, which means that this approach has no worst case guarantees. In essence, the problem is that we may be executing more range queries than the number of occurrences found.

Definition 11 *A* ***spurious*** *entry for string T in the suffix tree $\mathcal{T}$ is a leaf v of $\mathcal{T}$ such that v^R is a leaf of $\mathcal{T}^R$ and v is not a block in the $\mathcal{T}$-maximal parsing of T.*

For a dictionary $\mathcal{T}$ without spurious entries, we can guarantee that some orthogonal range queries must return occurrences.

Lemma 5 *Assuming $\mathcal{T}$ has no spurious entries for T and v is a leaf of $\mathcal{T}$, then the query $W(\textsc{Root}_{\mathcal{ST}}, v^R) : [0, t]$ returns at least one linking point.*

Proof There is some α such that $(\alpha.v)^R$ is a leaf in $\mathcal{T}^R$. Since $\mathcal{T}$ is a suffix tree and v is a leaf of $\mathcal{T}$, then $\alpha.v$ is also a leaf of $\mathcal{T}$. Hence, at least one linking point will be found by $W(\textsc{Root}_{\mathcal{ST}}, v^R) : [0, t]$, since $\textsc{Dfs}_{\mathcal{ST}}((\alpha.v)^R) \in W(\textsc{Root}_{\mathcal{ST}}, v^R)$. □

Spurious entries may be safely removed from the dictionary. Removing spurious entries can be done by considering $\mathcal{T}$ and $\mathcal{T}^R$ as a DAG, a node w in the DAG represents simultaneously v and v^R; there is an edge from w to w' if that edge exists in $\mathcal{T}$ or in $\mathcal{T}^R$. To remove spurious entries we perform a DFS over this DAG. We remove nodes that do not have blocks and are sinks or unary and the edge comes from $\mathcal{T}$. The nodes are checked and removed in their finishing time (see Cormen et al. [4] for definitions). This procedure runs in $O(d)$ time. Note that the resulting structure remains a suffix tree.

4.3 Occurrences Spanning more than a Single Block

In this section we focus on finding occurrences that span two or more consecutive blocks, i.e. type > 1. The ideas presented in this section are similar to those of Kärkkäinen et al. [14] and related with the approach proposed by Ferragina et al. [5].

We are now faced with the problem of retrieving the words in our dictionary that appear concatenated in $\mathcal{T}(T)$ and have P as a substring. Suppose that $P = cbdbddc$ and that we split P in two as $cbdbdd$ and c. We will now search for c in $\mathcal{T}$ and for $cbdbdd$ in $\mathcal{ST}$. The point c in $\mathcal{T}$ induces the range $I(c) = [5, 8]$; on the other hand, string $cbdbdd$ is parsed into cbd, bd, b and hence will be translated into 748. To search on the sparse suffix tree, we need $R(748) = 6'7'8'$. This will induce the range $[5^*, 5^*]$. Finally, to solve our problem we perform the orthogonal range query $[5^*, 5^*] : [5, 8]$ over the linking points $\mathcal{L}$. This corresponds to the question: is the string *cbdbdd*, parsed as *cbd.bd.d*, ever followed by a block that starts by c? The answer is yes, since there is a linking point in $[5^*, 5^*] : [5, 8]$. This point corresponds to *cbd.bd.d.cba.ba.ba*.

Observe that in this procedure we are using one suffix tree ($\mathcal{T}$) in the usual way, to search the text from right to left, and another suffix tree ($\mathcal{ST}$) to search the text in the opposite direction. Thus we are able to search for P by starting the search from the middle of the pattern. We "cross" the results, by using orthogonal range queries, to obtain the occurrences of P.

We will now explain how to determine in which points to break P. The pattern should be separated in the head and tail of $P[i..]$, for every $0 < i < m$, to account for every possible translation that can occur. These points can be located using the following dynamic programming equations:

$$tail(P[i..]) = \begin{cases} trace\_right[i] & \text{, if } |trace\_right[i]| = m - i \\ tail(P[i + |father\_right[i]|..]) & \text{, otherwise} \end{cases} \tag{1}$$

$$H(P[i..]) = \begin{cases} \epsilon & \text{, if } |trace\_right[i]| = m - i \\ father\_right[i].H(P[i + |father\_right[i]|..]) & \text{, otherwise} \end{cases}$$

We use Algorithm 2 to locate points $R(H(P[i..]))$ in $\mathcal{ST}$. We can use a similar procedure to compute $tail(P[i..])$. Whenever it is not possible to descend by a letter, the $\text{DESCEND}_{\mathcal{ST}}$ procedure returns the $udef$ state. See table 3 (bottom) for an example of this computation. Assume that the descend and suffix walk of P is already computed. Hence, the arguments of $\text{DESCEND}_{\mathcal{ST}}$ are available when $\text{DESCEND}_{\mathcal{ST}}$ is executed. Therefore, Algorithm 2 runs in $O((m/\epsilon) \log d)$ time, since it runs m times the $\text{DESCEND}_{\mathcal{ST}}$ operation, which requires $O((\log d)/\epsilon)$ time.

Notice the importance of using the $\mathcal{T}$-maximal parsing of T, instead of the original LZ78 parsing. By using a maximal parsing we have the guarantee that the notion of head is well defined. This means that to every $P[i..]$ we associate at most one point $R(H(P[i..]))$ in $\mathcal{ST}$. If we were using the original LZ78 parsing there could be $O(m)$ points in $\mathcal{ST}$ that corresponded to a given suffix $P[i..]$. Locating all those points would raise the overall complexity to $O(m^2)$.

Having located $tail(P[i..])$ in $\mathcal{T}$ and $R(H(P[i..]))$ in $\mathcal{ST}$, we know where to break the pattern. Now, all that we need are the ranges for the range query. The range for $\mathcal{T}$ is simply $I(tail(P[i..]))$. Whenever $P[..i-1]^R$ is a node of $\mathcal{T}^{\mathcal{R}}$, the range for $\mathcal{ST}$ is $W(R(H(P[i..])), P[..i-1]^R)$.

Let us consider, for example, the case of $i = 3$. We have that $H(P[3..]) = 48$ and $R(H(P[3..])) = 6'7'$. Hence $W(6'7', (cbd)^R) = [5^*, 5^*]$, since $8'$ is the only descendant of itself in $\mathcal{T}^R$. This means that, when we are extending *bd.d* to the left by prepending a word from our dictionary that

Algorithm 2 Locate $R(H(P[i..]))$ Algorithm

procedure *Locate_HPI*
  for $i \leftarrow m-1, 0 < i$ **do**
    $R(H(P[i..])) \leftarrow \textsc{Root}_{\mathcal{ST}}$
    if $|\text{trace\_right}[i]| < m - i$ **then**
      $R(H(P[i..])) \leftarrow \textsc{Descend}_{\mathcal{ST}}(R(H(P[i+|father\_right[i]|..])), father\_right[i])$
    end if
  end for
end procedure

terminates in *cbd*, the only such word is *cbd*. Therefore, we end up considering only the node *cbd.bd.d*.

Our algorithm for finding type > 1 occurrences of P proceeds as follows:

- Compute the descend and suffix walk of P in $\mathcal{T}$.
- Compute $tail(P[i..])$ from the descend and suffix walk of P.
- Locate the $R(H(P[i..]))$ points in $\mathcal{ST}$ (see Algorithm 2).
- If $|father\_left[i]| = i$ then $P[..i-1]^R = R(father\_left[i])$, compute $W(R(H(P[i..])), R(father\_left[i]))$.
- Compute $I(tail(P[i..]))$ from $tail(P[i..])$ (see Lemma 4 and Equation 1).
- Compute the orthogonal range queries $W(R(H(P[i..])), R(father\_left[i])) : I(tail(P[i..]))$.

An example of our algorithm is shown in Table 3 (bottom). The only range query that finds occurrences (occ') is the $[5^*, 5^*] : [5, 8]$ query, as we have explained in this section.

Theorem 3 *This procedure is correct and complete.*

Proof (Correct) Algorithm 2 locates points $R(H(P[i..]))$ points in $\mathcal{ST}$. These points correspond to substrings $P[i..j]$ of P. The weak descents extend this substrings into prefixes $P[..j]$. Then $tail(P[i..])$ gives the corresponding suffix $P[j+1..]$. The orthogonal range query "crosses" these information and returns positions where the string $P[..j]$ is followed by $P[j+1..]$. Hence P occurs in these positions. (Complete) Suppose that P occurs in $T = v_1 \dots v_f$ in the blocks $v_j.v_{j+1}\dots$. Hence there is some i such that $P[i..]$ is a prefix of $v_{j+1}\dots v_f$ and $P = father\_left[i].P[i..]$ (see Algorithm 1 and Definition 5). The strings corresponding to $father\_left[i]$ and head of $P[i..]$ are contained in the range $W(R(H(P[i..])), R(father\_left[i]))$ and the strings corresponding to $tail(P[i..])$ in the range $I(tail(P[i..]))$. Therefore, P is found by the orthogonal range query. □

5 A Compressed Self-Index based on LZ78 Dictionaries

We found it interesting to present this work in a general form, since it seems relevant to explore other techniques for inferring dictionaries, given a text T. We will now give a concrete instantiation of the above algorithm, using the Ziv-Lempel 78 algorithm [29].

Definition 12 *The **LZ78 parsing** of a string T is the sequence $Z_1, \dots, Z_n$ of strings such that $T = Z_1 \dots Z_n$ and for every i, $Z_i = Z_j c$ where Z_j is the largest prefix of $Z_i \dots Z_n$ among the $Z_1, \dots, Z_{i-1}$.*

The strings $Z_1 \dots Z_n$ are referred to as blocks. Given a string T, we proceed as follows: compute the LZ78 parsing of $T^R = Z_1 \dots Z_n$, then consider the suffix tree for strings $\{Z_1^R, \dots, Z_n^R\}$ as our dictionary, denoted by $\mathcal{T}_{78}$. In our example T^R is parsed into $a, b, ab, abc, d, db, dbc$ and the resulting dictionary can be seen in Figure 1 (top-right). The following lemmas expose why the dictionary we propose is adequate in terms of space.

Lemma 6 *If the number of blocks of the LZ78 parsing of T is n then $\mathcal{T}_{78}$ has at most $2n$ nodes, i.e. $d \leq 2n$.*

Proof Observe that every suffix of a Z_i^R is a Z_j^R for some j. Therefore the set $\{Z_1^R, \ldots, Z_n^R\}$ is suffix closed. Hence a suffix tree based on $\{Z_1^R, \ldots, Z_n^R\}$ will have at most $2n$ nodes. □

Lemma 7 *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$-maximal parsing of T has at most n blocks, i.e. $f \leq n$.*

Proof The idea is to show that if a block v_i of the $\mathcal{T}_{78}$-maximal parsing is a substring of some Z_j^R then it is a suffix. Suppose that v_i is a substring of Z_j^R. We have that $Z_j^R = \alpha.v_i.\beta$. Since the dictionary is a suffix tree and Z_j^R is a node, $v_i\beta$ is also a node and hence a dictionary word. Since the parsing is maximal, we have that $v_i.\beta = v_i$, i.e. that v_i is a suffix of Z_j^R. □

5.1 Space and Time Complexity

With the previous results we will now determine the space and time complexity of our algorithm using an LZ78 dictionary.

Lemma 8 *The* Dfs'$_{78}$ *operation can be supported over $\mathcal{T}_{78}$ in $O(1)$ time with $o(u \log \sigma)$ bits.*

Proof This result is obtained from lemma 4. Observe that t, the number of points of $\mathcal{T}_{78}$, can be at most u. Moreover the largest value of uH_0 can be at most $2u/\log_\sigma u$ since the number of 1's in the bitmap is at most $2n$ and Ziv et al. [29] showed that $n \leq (u/\log u)\log\sigma$. A few calculations show that the space occupied by this bitmap is at most $2u\log\sigma(\log\log u/\log u) + o(u\log\log u/\log u)$ bit, which is $o(u\log\sigma)$. □

We will refer to the index that uses LZ78 dictionaries as the Inverted-LZ-Index. The next theorem gives an overview of the space/time complexity of this structure. A previous version of this result [25], required more space.

Theorem 4 *Let d and d' be the number of nodes of $\mathcal{T}_{78}$ and $\mathcal{ST}_{78}$ respectively. Let t be the number of points of $\mathcal{T}_{78}$. Let f be the size of the $\mathcal{T}_{78}$-maximal parsing of T. The space/time trade-off of the Inverted-LZ-Index can be summarized as follows:*

Space in bits	$[\frac{d}{n}(1+\epsilon) + \frac{d'}{n}(1+\epsilon) + \frac{f}{n}]uH_k + o(u\log\sigma)$ $\leq (5+4\epsilon)H_k + o(u\log\sigma)$
Time to count	$O((occ + m/\epsilon)\log n)$
Time to locate	*free after counting*
Time to display l chars	$O(l/\epsilon)$, *improvable to* $O(l/(\epsilon\log_\sigma u))$ *with u extra bits*
Conditions	$k = o(\log_\sigma u)$, $\sigma = O(n)$, $0 < \epsilon \leq 1$, ϵ *is constant*

Proof (Space) The space requirements come from adding up the space of $\mathcal{T}_{78}$, $\mathcal{ST}_{78}$ and the range data structure. The $\mathcal{T}_{78}$ suffix tree requires at most $(1+\epsilon)d\log d + 5d + o(d)$, according to lemma 3. Moreover, to support Dfs'$_{78}$ we need $o(u\log\sigma)$ extra bits. The $\mathcal{ST}_{78}$ sparse suffix tree requires $(1+\epsilon)d'\log d' + 5d' + o(d')$ bits, according to lemmas 3. The range data structure (wavelet tree) requires another $f\log f(1+o(1))$ bits. The dominant factors are the ones associated with $\log u$. According to lemmas 6 and 7 these are the factors of $\log d$, $\log d'$ and $\log f$. Hence the overall log factor is $d(1+\epsilon) + d'(1+\epsilon) + f$. Ziv et al. [29] showed that $\sqrt{u} \leq n \leq u/\log_\sigma u$, and, therefore $n = o(u\log\sigma)$, which means that all remaining requirements are $o(u\log\sigma)$. The relation between n and H_k was established by Kosaraju et al. [15] who showed that $n\log u = uH_k + o(u\log\sigma)$ for $k = o(\log_\sigma u)$. Therefore, the expression in the theorem accounts for the space requirements of the ILZI.

(Count/Locate) We have already seen that Algorithm 1 runs in $O((m/\epsilon)\log\sigma)$ time. The time to find occurrences of type 1 is $O((1+occ_1)\log n)$. Observe that the number of queries computed is less than or equal to twice the number of leaves below P. By lemma 5 we know that the queries at the leaves must return occurrences. Therefore the total time amortizes to $O((1+occ_1)\log n)$. The time to find occurrences of type > 1 is the time of Algorithm 2, plus m weak descents and

m range queries. Therefore the total time for occurrences of type > 1 is $O((occ_{>1} + m/\epsilon)\log n)$, where $occ_{>1}$ is the number of type > 1 occurrences.

(Display) Observe that even though we do not store $R(\mathcal{T}_{78}(T))$ explicitly, we have $O(1/\epsilon)$ access time to it. The idea is to store a pointer to the leaf of $\mathcal{ST}_{78}$ with path-label $R(\mathcal{T}_{78}(T))$, denoted by $\text{FirstLeaf}_{\mathcal{ST}}$. Therefore $R(\mathcal{T}_{78}(T))[i] = \text{Letter}_{ST}(\text{FirstLeaf}_{\mathcal{ST}}, i)$. Hence, we can compute the j-th letter of $R(\mathcal{T}_{78}(T))[i]$ as $\text{Letter}(\text{Letter}_{ST}(\text{FirstLeaf}_{\mathcal{ST}}, i), j)$, in $O(1/\epsilon)$ time. To achieve optimal $O(l/(\epsilon \log_\sigma u))$ time we use an approach based on the work of Sadakane [28], similar to Arroyuelo et al. [1]. We define a new bitmap D', similar to bitmap D, used to retrieve the first $\log u$ bits of a node v instead of the first letter. This requires $d + o(d)$ bits. We also need a bitmap Q that indicates which sequences of $\log u/2$ bits do appear as the first bits of some v. By $(i)_2$ we denote the binary representation of i, with $\log u/2$ bits. The Q bitmap is defined as $Q[i] = 1$ iff $(i)_2$ is the prefix of some $(v)_2$ padded with zeros. Bitmap Q contains $2^{\log u/2} = \sqrt{u}$ bits and can therefore be stored in $o(u)$ bits. With these bitmaps we are able to retrieve $\log u/2$ bits from a block in $O(1)$ time, i.e. $\log_\sigma u/2$ letters. We repeat these bitmaps for $\mathcal{ST}_{78}$ and hence are able to retrieve $\log u/2$ bits from consecutive blocks. Finally we need another bitmap to be able to skip blocks. We use a bitmap V that marks the beginnings of the blocks in $R(\mathcal{T}_{78}(T))$. This requires $u + o(u)$ bits. As pointed out by Arroyuelo et al. [1], this bitmap can be used to report the occurrences of P as positions in T instead of as a block and an offset. □

The worst-case of the space expression is $(5 + 4\epsilon)H_k + o(u \log \sigma)$. However the worst example we were able to find, based on De Bruijn cycles, yielded $(4 + 3\epsilon)H_k + o(u \log \sigma)$ bits. In the next section we show concrete values for the space expression.

Finally note that the bound by Kosaraju et al. [15] concerns $H_k(T)$ not $H_k(T^R)$. This makes little difference. In theory Ferragina et al. [5] showed that $uH_k(T) - O(\log u) \leq uH_k(T^R) \leq uH_k(T) + O(\log u)$. In practice, $H_k(T)$ and $H_k(T^R)$ can also be shown to be similar. Moreover, we can switch the roles of T and T^R in our approach and search for P^R instead of P. In fact our prototype works precisely in this way. However we believe this would have made the exposition more complex and it would make it harder to point out the importance of the $\mathcal{T^R}_{78}$ suffix tree.

6 Practical Issues and Testing

6.1 Practical considerations

We implemented a prototype to test these ideas. Navarro [21] pointed out that, by using a naive search instead of the range data structure, it was possible to build a smaller index that was faster in practice. The naive way to compute an orthogonal range query is to choose the smallest range and, for each point of that range, check whether the point belongs to the other range. Suppose, for example, we wish to compute the range query $W(\text{Root}_{\mathcal{ST}}, 2') = [1^*, 4^*] : [0, 9] = [0, t-1]$. First, observe that, when we refer to the smallest range, we are referring to the range in the $[1, f] \times [1, f]$ grid not in the $[0, d'-1] \times [0, t-1]$ space. Therefore we reduce the $[1^*, 4^*] : [0, 9]$ query to the $[1^{p*}, 3^{p*}] : [1^p, 6^p]$ query. Obviously, the smallest range is $[1^{p*}, 3^{p*}]$. Since, for this particular query, the second range covers the whole space, the result is $[1^{p*}, 3^{p*}]$, which corresponds to $\{2^*, 3^*, 4^*\}$. We have already seen that this type of queries is used for type 1 occurrences. Therefore, using this method, the time to compute the range queries for type 1 occurrences is $O(occ_1)$. For type > 1 occurrences this procedure has no worst case guarantees. However, in practice, this is acceptable and more efficient. Therefore we did not implement the range data structure and we used this approach instead. This immediately removes our capability of reducing $[0, t-1]$ to $[1, f]$, which means that we cannot use points of $\mathcal{T}$ to support the linking points. This means that there is no reason to use a compressed bitmap to support the Dfs' operation for points that are not nodes, as described in lemma 4. Instead we store $\langle \text{Dfs}(R(V(v_1 \ldots v_i))), \text{Dfs}(v_{i+1})\rangle$ when $i < f$ and $\langle \text{Dfs}(R(V(v_1 \ldots v_i))), 0\rangle$ when $i = f$, since v_{i+1} is the largest prefix of $v_{i+1} \ldots v_f$ that is a node in $\mathcal{T}$. Observe that the linking points in our example actually coincide exactly with this definition, (see Figure 1 bottom-right). To find the linking points associated with a node v of

$\mathcal{T}$, we find the leaves below point $R(v)$ in $\mathcal{ST}$. Moreover, to decide which range is smaller, we estimate the number of points in $I'(v)$ as the number of points in $W(\text{ROOT}_{\mathcal{ST}}, R(v))$. In Navarro's approach, occurrences of type > 1 are further distinguished between type 2 and type > 2. Navarro did not use dynamic programming, because it is possible to guarantee that there are not too many occurrences of type > 2. Type > 2 occurrences span more than two blocks. The fundamental argument is that, since the LZ78-blocks are all distinct, a given Z_i occurs in at most one position. Therefore the $P[i..j]$ substrings of P occur in at most $O(m^2)$ positions. Hence there cannot be more than $O(m^2)$ type > 2 occurrences of P in the LZ78 parsing of T. For $\mathcal{T}(T)$ no such result exists. However, even though a word v may correspond to more than one block of $\mathcal{T}(T)$, in average it does not correspond to many. Therefore we do not use dynamic programming either. Instead, we use different procedures for type 2 and type > 2 occurrences.

There is not a very compelling reason to store $\mathcal{ST}_{78}$ as a suffix tree when not using dynamic programming. Inverted files store a list of occurrences for every dictionary word. These lists are usually ordered by the position in T of the occurrences of the words. This regularity is usually explored, for example, with delta coding, to store these lists in compressed form. This property is also important when searching for patterns because, since the type > 1 search scans the text sequentially, it provides better cache performance. Our implementation of $\mathcal{ST}_{78}$ is similar to a sparse suffix array, i.e. a suffix array for $R(\mathcal{T}(T))$. However, the suffixes of $R(\mathcal{T}(T))$ are only sorted by the first block. Suffixes that start with the same block are ordered by position in $R(\mathcal{T}(T))$, just like in inverted files.

A very important aspect of our prototype is that the implementation of $\mathcal{T}_{78}$ differs considerably from the succinct representation we presented. The fundamental reason for this fact is that the succinct implementation would suffer from poor cache performance. Instead we opted for a more cache aware implementation. The $\mathcal{T}_{78}$ tree is implemented in a pointer like fashion. Every node is stored in a memory cell indexed by its breath-first time-stamp. For example, node cb will be stored in cell 3. The LETTER operation is replaced by a HEAD pointer, that, for every node v with father node $v[..i-1]$, points to node $v[i..]$. This information suffices to read edge-labels, by using suffix links. Every node v stores a CHILD pointer, its DFS time, a suffix link, the string depth, the HEAD pointer and pointers indicating $W(\text{ROOT}_{\mathcal{ST}_{78}}, v^R)$ over $\mathcal{T}_{78}$. This provides better cache performance in several points. First, we store the information in the nodes and the topological structure of the tree together. Second, there is no need to traverse back and forth from $\mathcal{T}_{78}$ to $\mathcal{T}^{\mathcal{R}}{}_{78}$ to read edge-labels or compute suffix links. Third, the BFS ordering avoids some cache faults in branching. Clearly, implementing $\mathcal{T}_{78}$ this way requires more space than the succinct implementation. This constitutes a severe problem. In order to solve it, we infer a smaller dictionary, i.e. a $\mathcal{T}_{78}$ tree with less nodes. In practice, we use the following variation of the LZ78 parsing:

Definition 13 *The **LZ78 parsing with quorum l** of a string T is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$ and, for every i, $Z_i = Z_j c$ where c is a letter and Z_j is the largest prefix of $Z_i \ldots Z_n$ that appears at least $l+1$ times among the $Z_1, \ldots, Z_{i-1}$.*

Clearly the LZ78 parsing with quorum 0 corresponds to the usual notion of LZ78 parsing. In practice a quorum of 2 compensates for the space requirements of $\mathcal{T}_{78}$ without affecting performance too much. Table 4 shows the size of the ILZI for different quorum values. Variable i represents the size of different indexes, in bits. Therefore $i/2^{23}$ is the size in megabytes, $i/8u$ is the ratio with respect to the size of the original string and i/uH_k is the ratio with respect to the size of the compressed string. Our results show that increasing the quorum value significantly reduces the space requirements of the ILZI while degrading the time performance only slightly. Observe that with a quorum of 2 our index has acceptable space requirements, in practice. Our results also show the ILZI has acceptable space requirements in theory. For example the results show that for the xml file the practical value is $2.65uH_k$ bits and the theoretical value is $(2.49 + 1.62\epsilon)uH_k + o(u \log \sigma)$ bits.

	sources.50*MB*			*dblp.xml*.50*MB*			*dna*.50*MB*			*proteins*			*pitches*			*english*.50*MB*		
l	$i/2^{23}$	$i/8u$	i/uH_k	$i/2^{23}$	$i/8u$	i/uH_k	$i/2^{23}$	$i/8u$	i/uH_k	$i/2^{23}$	$i/8u$	i/uH_k	$i/2^{23}$	$i/8u$	i/uH_k	$i/2^{23}$	$i/8u$	i/uH_k
32	50.0	1.00	2.80	26.9	0.54	2.73	30.9	0.62	2.24	85.9	1.35	2.54	75.8	1.42	2.83	47.6	0.95	2.63
16	48.0	0.96	2.69	24.8	0.50	2.52	31.3	0.63	2.27	85.6	1.34	2.53	73.9	1.39	2.76	46.4	0.93	2.56
8	46.6	0.93	2.61	24.2	0.48	2.46	33.0	0.66	2.39	87.5	1.37	2.59	72.6	1.36	2.71	45.8	0.92	2.53
4	48.6	0.97	2.72	24.1	0.48	2.45	37.0	0.74	2.68	93.7	1.47	2.77	76.4	1.43	2.85	48.5	0.97	2.68
2	53.5	1.07	3.00	26.1	0.52	2.65	44.0	0.88	3.19	102.8	1.61	3.04	84.7	1.59	3.16	54.3	1.09	2.99
1	59.6	1.19	3.34	28.9	0.58	2.93	52.5	1.05	3.81	120.4	1.89	3.56	97.9	1.84	3.65	61.8	1.24	3.41
0	87.4	1.75	4.90	42.5	0.85	4.32	92.6	1.85	6.72	226.5	3.55	6.69	161.7	3.04	6.04	93.3	1.87	5.15
	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n	d/n	d'/n	f/n
	0.60	1.19	0.90	0.54	1.08	0.87	0.92	1.20	0.97	0.85	1.22	0.98	0.76	1.25	0.94	0.64	1.33	0.94
	total			*total*			*total*			*total*			*total*			*total*		
	2.70	+	1.79ϵ	2.49	+	1.62ϵ	3.09	+	2.12ϵ	3.04	+	2.07ϵ	2.95	+	2.01ϵ	2.90	+	1.96ϵ

Table 4 Space requirements of the ILZI index for different quorum values. Variable l represents different quorum values. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/8u$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/u$. The bottom part of the column shows empirical values for the following ratios d/n, d'/n and f/n. Note that these values can be used to determine factor of uH_k associated with the ILZI, we present this value below total (see Theorem 4). Observe that the factor associated with the ϵ corresponds to removing the range data structure.

6.2 Experimental Results

We compared our implementation the Inverted-Lempel-Ziv-Index (ILZI), against the implementations provided in the Pizza&Chili corpus [22][3]. As texts, we used the files in the Pizza&Chili corpus, with approximately 50 Megabytes each. The indexes were parametrized to occupy approximately the same space whenever possible. The indexes used were the following: Raw is the raw string, ILZI is the inverted-Lempel-Ziv-index using the algorithm described in this paper, LZI Navarro's LZ-index, LZI-1 is the improvement of the LZI by Arroyuelo et al., NFMI is an implementation of the FM-index by Navarro, CSAx8 is Sadakane's compressed suffix array, SSA is the succinct suffix array, RL is the run-length FM-index, AFFMI is the alphabet friendly FM-index, FMI2 is the second version of the FM-index and SAC is the suffix array in uncompressed form, packed in bits. We omitted the compressed compact suffix array, because it was not competitive. We also omitted the suffix array packed in words because it was very similar to SAC.

In table 5 we show the space requirements of different compressed indexes for the sample files. The *par* line gives the parameters used for indexes that require it. The parameters were chosen so that the resulting index occupied approximately the same size as the ILZI. However, some minimal values were used for performance reasons. For the CSArray we give the D value, for CSAx8 we have that $L = 8 \times D$. Figures 4,5,6,7 show the time performance of different compressed indexes. The performance of compressed indexes can be described as $\Theta(m.C + occ.R + out.O)$, where *out* is the number of letters that we wish to display, C is the counting factor, R is the reporting factor and O is the outputting factor. For some compressed indexes it is possible to run the indexes in counting mode and the resulting time is $\Theta(m.C)$. However for Lempel-Ziv indexes this is not possible, and our index runs in $\Theta(m.C + occ.R)$ even for counting, albeit with a smaller R constant. We determined the factors and overall query time for all the indexes. We show the results for different values of m in figures 4,5,6,7. To obtain these results we ran tests of 60 seconds each with a minimal number of 5 repetitions. For counting, this means that we tested from 6×10^4 to 6×10^8 patterns. For reporting, we tested from 5 to 6×10^6 patterns and each pattern had at least one occurrence. For outputting, we displayed 60 characters per occurrence.

The fact that LZ-based indexes cannot operate in counting mode can be observed empirically since the time of these indexes is not constant in the time to count graphs. As expected, when m increases *occ* decreases and the time also decreases. Eventually, the overall time becomes competitive with other compressed indexes. For most examples this happens when m is around 20.

[3] Tested on Pentium 4, 3.2 GHz, 1 MB of L2, 1Gb of RAM, with Fedora Core 3, compiled with `gcc-3.4 -O9`.

		Raw	ILZI	LZI	NFMI	CSAx8	LZI-1	SSA	RL	AFFMI	FMI2	SAC
sources.50MB	$i/2^{23}$	50.0	53.5	80.9	68.1	54.6	74.5	57.2	53.2	54.1	53.7	212.5
	$i/8u$	1.00	1.07	1.62	1.36	1.09	1.49	1.14	1.06	1.08	1.07	4.25
	i/uH_k	2.80	3.00	4.53	3.81	3.06	4.18	3.20	2.98	3.03	3.01	11.91
	par				5	7		512	26	36	0.17	
dblp.xml.50MB	$i/2^{23}$	50.0	26.1	44.5	64.9	25.8	40.9	54.6	33.6	32.9	28.7	212.5
	$i/8u$	1.00	0.52	0.89	1.30	0.52	0.82	1.09	0.67	0.66	0.57	4.25
	i/uH_k	5.08	2.65	4.52	6.60	2.62	4.16	5.55	3.42	3.35	2.91	21.60
	par				5	19		512	512	512	0.1	
dna.50MB	$i/2^{23}$	50.0	44.0	60.9	63.4	44.6	55.1	44.3	44.1	43.5	47.1	212.5
	$i/8u$	1.00	0.88	1.22	1.27	0.89	1.10	0.89	0.88	0.87	0.94	4.25
	i/uH_k	3.63	3.19	4.42	4.60	3.23	4.00	3.21	3.19	3.16	3.42	15.41
	par				5	11		24	64	24	0.17	
proteins	$i/2^{23}$	63.7	102.8	152.9	100.9	100.1	137.9	108.2	104.1		109.0	270.8
	$i/8u$	1.00	1.61	2.40	1.58	1.57	2.16	1.70	1.63		1.71	4.25
	i/uH_k	1.88	3.04	4.52	2.98	2.96	4.08	3.20	3.08		3.22	8.00
	par				10	6		10	12		0.27	
pitches	$i/2^{23}$	53.2	84.7	124.8	86.8	86.1	115.7	87.5	84.8		92.9	226.3
	$i/8u$	1.00	1.59	2.34	1.63	1.62	2.17	1.64	1.59		1.74	4.25
	i/uH_k	1.99	3.16	4.66	3.24	3.21	4.32	3.27	3.17		3.47	8.44
	par				9	5		16	12		0.26	
english.50MB	$i/2^{23}$	50.0	54.3	81.1	66.8	56.3	73.2	54.1	52.2	54.6	53.2	212.5
	$i/8u$	1.00	1.09	1.62	1.34	1.13	1.46	1.08	1.04	1.09	1.06	4.25
	i/uH_k	2.76	2.99	4.47	3.69	3.11	4.04	2.99	2.88	3.01	2.94	11.73
	par				5	7		64	30	24	0.17	

Table 5 Table with the size of different compressed indexes for sample files. It shows the space requirements of different indexes, the original string (Raw), the Inverted-LZ-Index (ILZI), Navarro's LZ-index (LZI), reduced LZI (LZI-1), Navarro's implementation of the FM-index (FMI), Sadakane's CSArray (CSAx8), the succinct suffix array (SSA), the compressed compact suffix array (CCSA), the run-length FM-index (RL), the alphabet friendly FM-index (AFFMI), the second version of the FM-index (FMI2), SAC is the suffix array in uncompressed form, packed in bits. Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/8u$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/u$. The *par* line gives the parameters used for indexes that require it. For the CSArray we give the D value, for CSAx8 we have that $L = 8 \times D$.

The counting graphs also show that reducing the dependency on m from $O(m^2)$ to $O(m)$ had significant impact in the query time. This makes our index up to an order of magnitude faster than LZI and LZI-1 for counting when m is large. On the contrary, for small patterns ($m = 5$) it is up to 2.6 times slower than LZI and up to four orders of magnitude slower than the other compressed indexes.

On the other hand LZ-based indexes are extremely fast at reporting occurrences. In fact they are the only self-indexes using $O(uH_k)$ bits able to spend $O(\log n)$ time per occurrence in practice. This is also visible in the graphs since the reporting factor of LZ-based indexes is around an order of magnitude smaller than that of other compressed indexes.

The displaying time per character is not a very decisive factor to tell indexes apart since all of them are very fast. The FM-index performed extremely well on natural language based files. The LZ-based indexes had more stable performance and are among the fastest for all samples. The suffix arrays are around two orders of magnitude faster than the compressed indexes, most likely due to cache effects

7 Conclusions

This paper presents two fundamental observations on LZ78 based compressed indexes. The first one is that the tree ($\mathcal{T}_{78}$) build with the reverse blocks of the LZ78 parsing is a suffix tree. This structure was first presented by Kärkkäinen [13], but this version required T to be present and since it was based in LZ77, it was not necessarily a suffix tree. In the work presented by Navarro [21] the structure is called RevTrie, but its suffix tree nature is not explored and, in fact, reading an edge-label requires $O(m^2)$. In the work presented by Ferragina and Manzini [5] it appears as an FM-Index of $T_\$^R$. They present a proof that its space requirements can be related to the entropy of the text T, in a different way from us. Moreover, its suffix tree structure is also not explored. This observation is fundamental for our approach since it allows us to compute a descend and suffix walk instead of having to search for all the substrings of P. The second observation is about the way the same string appears in the LZ78 parsing. A string S may appear in $O(m)$ different ways as the concatenation of LZ78 blocks. This, in turn, forces algorithms based on the LZ78 parsing to have quadratic behavior. We solve this problem by discarding the original parsing and using a maximal parsing. In the maximal parsing, a string S appears in at most one way as the concatenation of blocks. Navarro uses the original LZ78 parsing. Ferragina and Manzini discard the parsing and solve the problem by using an FM-index, i.e. resorting to the Burrows-Wheeler transformation.

Our index is a significant contribution to LZ-based compressed indexes. We improved the counting time performance of LZ-based indexes to linear time on m. At the same time, the structure we propose is smaller than LZI, for all the files we tested. In practice, with the terms we obtained in table 4, we can choose an ϵ to make the index smaller than $4uH_k + o(u \log \sigma)$. In fact it can be seen in table 4 that our implementation of the ILZI is always smaller than the LZI. However a new version of the LZ-index proposed by Arroyuelo et al. [1] requires only $(2+\epsilon)uH_k + o(u \log \sigma)$ with worst case guarantees. Without worst case guarantees it requires $(1+\epsilon)uH_k + o(u \log \sigma)$ bits and it has $O(m^2)$ average search time for $m \geq 2 \log_\sigma u$. It is interesting to notice that Arroyuelo et al. independently explored the suffix tree structure of $\mathcal{T}_{78}$ to reduce the time to read an edge-label to $O(m)$. We cannot achieve the reduced space requirements of Arroyuelo et al. essentially because we are storing more structures. In fact, as a second contribution of this paper, we pointed out a possible representation of suffix trees (lemma 3). This representation is not very competitive when compared to the compressed suffix trees presented by Sadakane [26]. Nevertheless, it is adequate for our goals. For suffix trees, in general, it requires more space than the representation of Sadakane. In fact, the problem is the space required to store R and R^{-1}, $(1+\epsilon)n \log n$ bits. Arroyuelo et al. [1] showed how to reduce the space requirements of R. However, even with such an improvement, it is still not comparable to Sadakane's approach in terms of space. We expect further work based on this approach to produce a competitive representation.

Acknowledgments

We are deeply grateful to Gonzalo Navarro for several reasons: organizing the Workshop on Compression, Text, and Algorithms at DCC in November of 2005, that motivated stimulating discussions on compressed indexes; providing prototypes together with Kunihiko Sadakane, Rodrigo Gonzalez, Paolo Ferragina, Giovanni Manzini, Rossano Venturini, Veli Mäkinen; creating the Pizza&Chili Corpus together with Ferragina; for suggestions and corrections along with Diego Arroyuelo and several anonymous reviewers. We would like to thank Luis Coelho for countless discussions about our this index.

References

1. D. Arroyuelo, G. Navarro, K. Sadakane. Reducing the space requirement of LZ-index. In Proceedings of *CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 318–329, 2006.
2. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
3. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. McGraw, second edition, 2001.

5. P. Ferragina, G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
6. P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro. An alphabet-friendly fm-index. In Proceedings of *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160, 2004, To appear in ACM TALG, 2007.
7. R. F. Geary, R. Raman, V. Raman. Succinct ordinal trees with level-ancestor queries. In Proceedings of *SODA*, pages 1–10, 2004.
8. S. Grabowski, G. Navarro, R. Przywarski, A. Salinger, V. Mäkinen. A simple alphabet-independent fm-index. *Int. J. Found. Comput. Sci.*, 17(6):1365–1384, 2006.
9. R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. In Proceedings of *SODA*, pages 841–850, 2003.
10. R. Grossi, J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.
11. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1999.
12. G. Jacobson. Space-efficient static trees and graphs. In Proceedings of *FOCS*, pages 549–554, 1989.
13. J. Kärkkäinen, E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In Proceedings of *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155, 1996.
14. J. Kärkkäinen, E. Ukkonen. Sparse suffix trees. In Proceedings of *COCOON*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230, 1996.
15. S. R. Kosaraju, G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999.
16. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
17. J. I. Munro. Tables. In Proceedings of *Proceedings of FSTTCS 1996*, volume 1180 of *LNCS*, pages 37–42, 1996.
18. J. I. Munro, R. Raman, V. Raman, S. S. Rao. Succinct representations of permutations. In Proceedings of *ICALP*, volume 2719 of *LNCS*, pages 345–356, 2003.
19. J. I. Munro, V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
20. G. Navarro, V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
21. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004.
22. G. Navarro, P. Ferragina. http://pizzachili.dcc.uchile.cl/.
23. R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In Proceedings of *ICALP*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604, 1999.
24. R. Raman, V. Raman, S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of *SODA*, pages 233–242, 2002.
25. L. M. S. Russo, A. L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. In Proceedings of *SPIRE*, volume 4209 of *Lecture Notes in Computer Science*, pages 163–180, 2006.
26. K. Sadakane. Compressed suffix trees with full functionality. to appear in Theory of Computing Systems.
27. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
28. K. Sadakane, R. Grossi. Squeezing succinct data structures into entropy bounds. In Proceedings of *SODA*, pages 1230–1239, 2006.
29. J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Fig. 4 Time results for counting. These graphs shows the impact of our improvement. This can be observed by comparing the ILZI and LZI indexes. The graphs also show the fact that LZ based indexes cannot count in optimal time. However they do become competitive when m increases, causing *occ* to decrease.

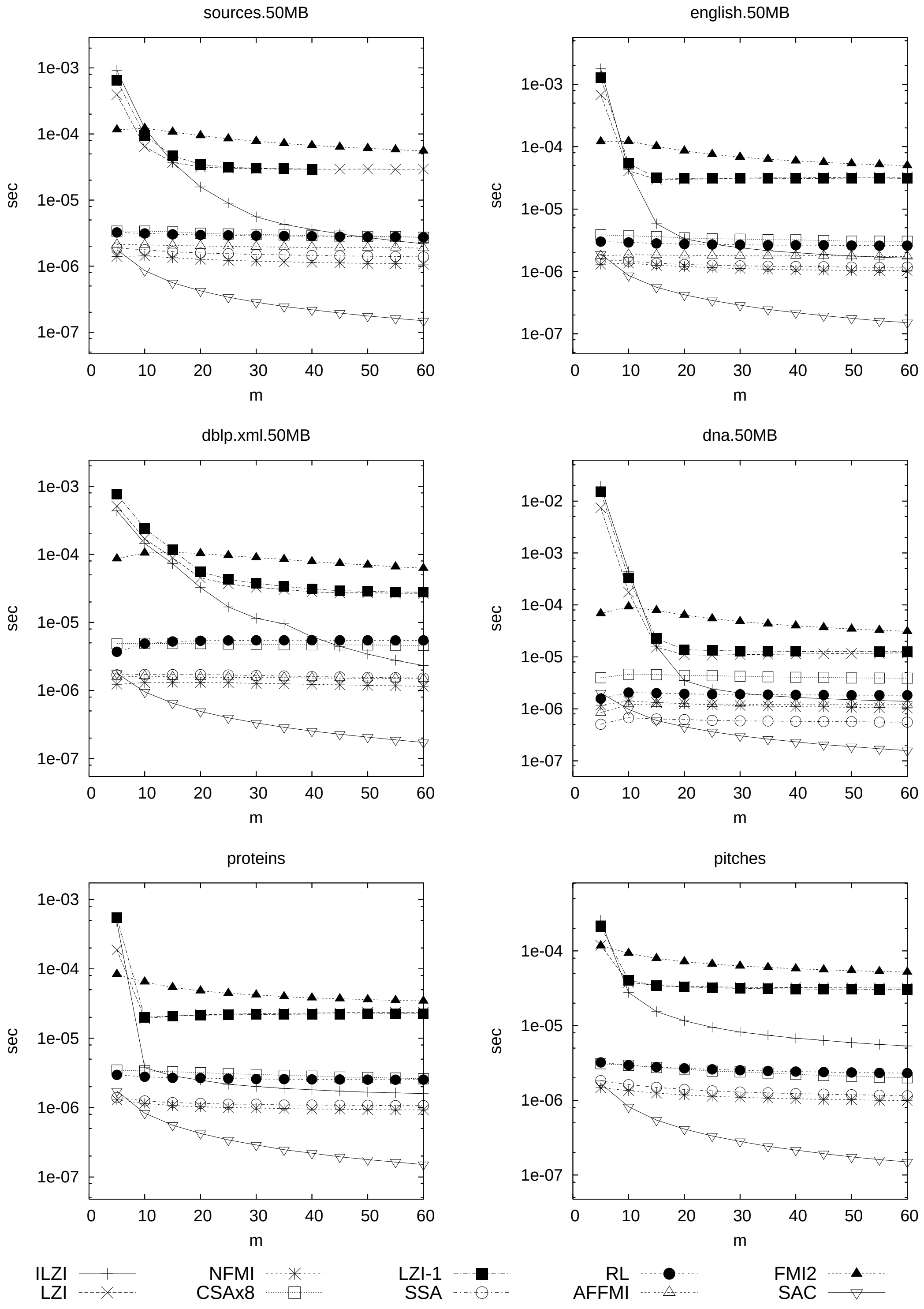

Fig. 5 Time results for reporting.

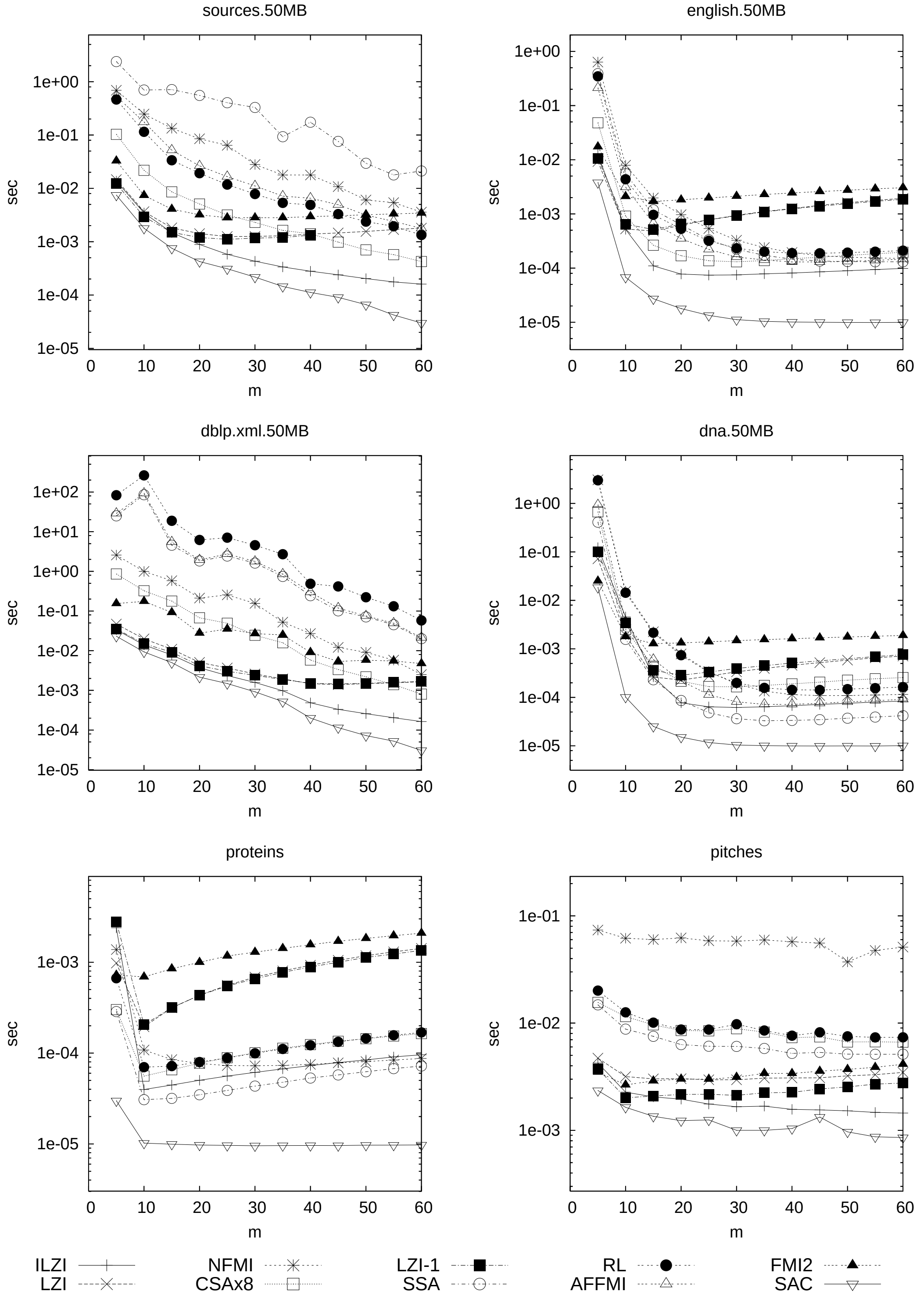

Fig. 6 Time results for reporting factor (R). This value is obtained by subtracting the counting time and dividing by the number of occurrences found. These graphs confirm that in fact LZ based indexes are the fastest at reporting occurrences. These results show that this factor is comparable to that of suffix arrays, being orders of magnitude faster than the alternatives.

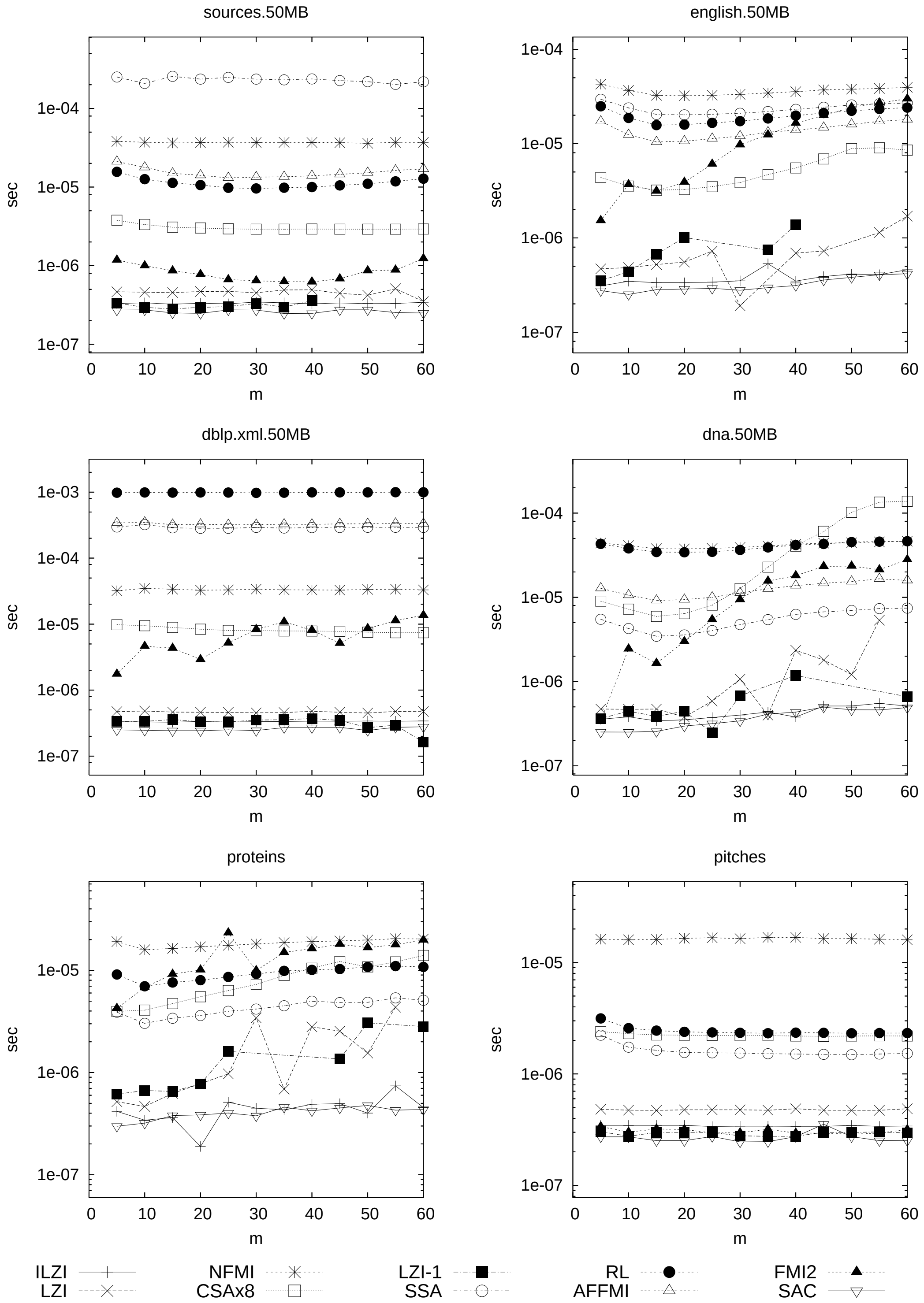

Fig. 7 Time results for outputting factor (O). These results show that the ILZI is among the fastest compressed indexes at outputting.