A Compressed Self-Index using a Ziv-Lempel Dictionary

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

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

Abstract. A compressed full-text self-index for a text T, of size u, is a data structure used to search patterns P, of size m, in T that requires reduced space, i.e. that depends on the empirical entropy (H_k, 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 $O((m + occ) \log n)$ time, where occ is the number of occurrences and σ the size of the alphabet of T. 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 the LZ-Index, the FM-index and a compressed suffix array.

1 Overview

The exact matching problem consists in searching for a short (pattern) sequence P in a longer (text) sequence T. Naive and linear solutions for this problem can be found in undergraduate computer science textbooks [1]. 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 stopped being 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 greedy. 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 less space demanding data structures have been proposed by several researchers [2–6]. Usually a text stored in compress format requires less space than its uncompressed version. 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. An important tool to describe the space of compressed indexes is the k-th order empirical entropy of T defined by Manzini [7], denoted simply by H_k. The 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 organise it globally. In a way our contribution is to organise globally Ziv-Lempel compressed indexes that were only locally organised. The empirical entropy provides a lower bound to the number of bits needed to compress T using a compressor that encodes

* 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

each character considering only the context of k characters that follow it in T. Makinen and Navarro presented a comprehensive survey on compressed full-text indexes [8].

A surprising way to reduce the space requirements of a full-text index, discovered in this line of research, is to turn it into a self-index. Basically it turned out that with a negligible amount of information, it is possible to make full-text indexes reproduce any substring of T without storing T explicitly.

Compressed suffix arrays [6, 2] and the FM-index [3] are the main trends of compressed indexes. This is partially due to the fact that LZ-indexes [3–5] 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. [5], which was not a self-index, required $O(m^2 + (m + occ) \log u)$ time and Navarro's [4] 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. [9]. 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. [3]. However this index requires a considerable amount of space, $O(uH_k(T) \log^{\epsilon} u) + o(u)$ bits. In fact the index presented by Ferragina et al. is not used in practice. Instead they simple add an FM-Index to their structure. Using an FM-Index may lead to alphabet related problems, i.e. large hidden σ dependencies. Some solutions have been presented to address this problem [10, 11]. However 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 organise T in an off-line way.

2 Basic Concepts and Notation

For basic concepts related to strings and suffix trees we refer the reader to Gusfield [12]. We use the following conventions: strings start at index position 0; prefixes, substrings and suffixes are denoted respectively as $S[..i]$, $S[i..j]$, $S[j..]$; 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 generalised suffix tree. The terminator symbols are not considered as part of the edge-labels. 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 v's suffix link. $\textsc{Letter}(v, i)$ 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. By $\textsc{Dfs}(v)$ we refer to the depth-first time-stamp [1] of a node v in a suffix tree and by $\textsc{Dfs}'(p)$ to the depth-first time-stamp of a point p in a suffix trie. As a running example consider $T = cbdbddcbababa$ and $\mathcal{T}$ as the suffix tree in figure 1 (top-right).

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

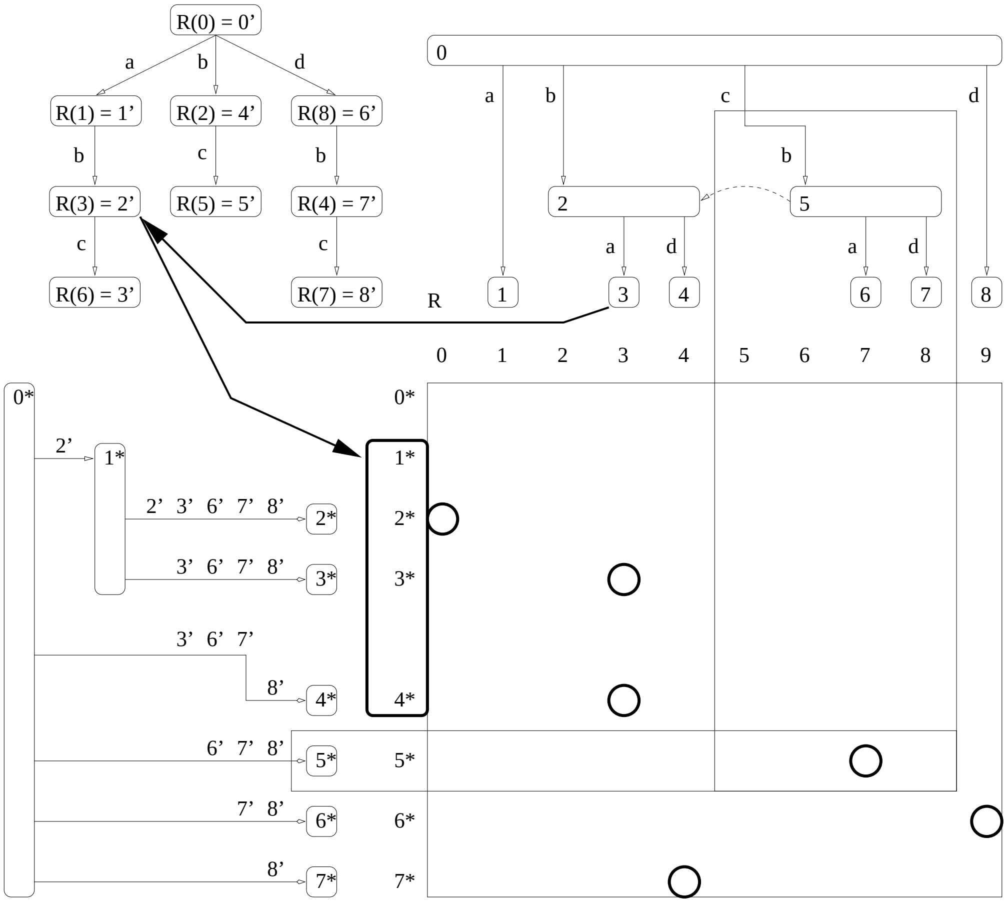

Fig. 1. (top-right) Suffix tree for strings $\{a, b, ba, bd, cba, cbd, d\}$. Suffix link from cb to b shown by a dashed arrow. Nodes show their DFS value in $\mathcal{T}$. (top-left) Reverse tree of the suffix tree on the right. Nodes show their DFS value in $\mathcal{T}^R$. The R mapping is shown and $R(3)$ is indicated by a bold arrow. (bottom-left) Sparse suffix tree of T, nodes show their 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 DFS$_{\mathcal{ST}}$ values. Orthogonal range query [5*,5*]:[5,8].

In our example DFS(c) is undefined, DFS$(cb) = 5$, DFS’$(c) = 5$, DFS’$(cb) = 6$, $I(c) = [5, 8]$.

Definition 2. *The* **reverse tree** $\mathcal{T}^R$ *of a suffix tree* $\mathcal{T}$ *is the minimal labelled 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 DFS(v) to DFS(v^R) (see figure 1). We will use $R(v)$ to denote $R(\text{DFS}(v))$. Note that since the nodes of $\mathcal{T}$ form a suffix closed set, the nodes of $\mathcal{T}^R$ form a prefix closed set.

2.1 Succinct Suffix Trees

Our approach is based on suffix trees. We start by presenting a representation of suffix trees that is adequate for our goals and analyse its space requirements.

By bitmap B we refer to a string over $\{0, 1\}$. 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 $\textsc{Select}(B,i)$ returns the smallest j such that $\textsc{Rank}(B,j+1)=i$. Munro [13] showed how to support these operations in $O(1)$ time and $|B|+o(|B|)$ bits.

Geary et al. [14] 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 $\textsc{Dfs}$ value among the descendants of v; $\textsc{Select}(j)$ returns the node with $\textsc{Dfs}$ 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.

We assume that the tree structure of $\mathcal{T}$ and $\mathcal{T}^R$ are stored using the previous representation. Arroyuelo et al. [9] 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. [15] 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 1. *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 $\textsc{Depth}_{\mathcal{T}^R}(R(v))$. 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 [2]. 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 $\textsc{Root}$, 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 bellow 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 $\textsc{Letter}$ 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 *Descend*?. $\square$

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

Lemma 2. *For a suffix tree $\mathcal{T}$ with d nodes and t points, operations $\textsc{Dfs'}(p)$ and $I(p)$ can be computed in $O(1)$ time using $(2+\lceil\log t\rceil-\lfloor\log d\rfloor)d+o(d)$ extra bits.*

Proof. Observe that the $\textsc{Dfs'}(v)$ values appear sorted in $\textsc{Dfs}(v)$ order. Therefore we can store the $\textsc{Dfs'}(v)$ values, for the nodes of $\mathcal{T}$, with the representation of Grossi et al. [6, Lemma 2]. 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. Also $I(p)=[\textsc{Dfs'}(p),\textsc{Dfs'}(\textsc{Select}(\textsc{RightRank}(v))]$. $\square$

2.2 Descend and Suffix Walks

Given a string P we can traverse a suffix tree $\mathcal{T}$ in greedy way, i.e. start at ROOT and descend as much as possible. When it is impossible to descend any further, follow suffix-links until descending becomes possible again, as in Algorithm 1.

Definition 3. *The* ***descend and suffix walk*** *of a string P over a suffix tree $\mathcal{T}$ is the sequence $p_0 \dots p_{2m}$ of points of $\mathcal{T}$ computed by Algorithm 1.*

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| 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 2. (Top) Descend and suffix walk of *cbdbddc* in $\mathcal{T}$. (Bottom) Values for locating type > 1 occurrences.

Definition 4. *The* ***right, left 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 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 node of $\mathcal{T}$.

Note that we define in an artificial way $\textsc{SuffixLink}(\textsc{Root})$ as a node that descends to the root by every letter including terminator symbols. It is important to notice that Algorithm 1 starts by appending to P a terminator character $\$'$ that fails to match with any other character. Observe that in Algorithm 1 the operation $\textsc{SuffixLink}$ is computed for points, not just nodes. This is done in the classical way. The operation $\textsc{SuffixLink}$ over points doesn't have $O(1/\epsilon)$ guaranteed time. However the total time of Algorithm 1 amortises to $O((m/\epsilon)\log\sigma)$ (see Gusfield [12] for details). Table 2 (top) shows the descend and suffix walk of *cbdbddc* in $\mathcal{T}$.

3 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.

3.1 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 organise 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 $v_1, \ldots, v_f$ such that $T = v_1 \ldots v_f$ 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*. 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 a string such that $V(v_1 \ldots v_f)[i] = \textsc{Dfs}(v_i)$.*

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 technical reasons we must reverse the string $\mathcal{T}(T)$. This is achieved by extending the canonical mapping R to sequences in the following way: $R(v_1 \ldots v_f) = R(v_f) \ldots R(v_1)$. 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***[1] *$\mathcal{ST}$ 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.

[1] Similar to a concept defined by Kärkkäinen et al. [16]

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* $\text{D}\textsc{fs}_{\mathcal{ST}}$ *values of the nodes below the following points:*
$\{p.\text{D}\textsc{fs}_{\mathcal{T}^R}(v') \mid v' \text{ is a descendant of } v^R \text{ in } \mathcal{T}^R\}$

For example, $W(\text{R}\textsc{oot}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, since this contains the $\text{D}\textsc{fs}_{\mathcal{ST}}$ values for the nodes below $2', 3'$ in $\mathcal{ST}$, see figure 1. This can be computed in $O((\log d)/\epsilon)$ time. We do two binary searches in the children of p, searching for $\text{L}\textsc{eft}\text{R}\textsc{ank}_{\mathcal{T}^R}(v)$ and $\text{R}\textsc{ight}\text{R}\textsc{ank}_{\mathcal{T}^R}(v)$. Then $W(p, v^R) = [\text{L}\textsc{eft}\text{R}\textsc{ank}_{\mathcal{ST}}(v''), \text{R}\textsc{ight}\text{R}\textsc{ank}_{\mathcal{ST}}(v''')]$, 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 $v_1, \ldots, v_i$ and string $v_{i+1} \ldots v_f$ such that $v_1, \ldots, v_i$ is the smallest sequence for which $v_{i+1} \ldots v_f$ 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.

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 $v_1 \ldots v_f$ of T is the following set:*

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

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

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

We need to process the linking points to be able to compute orthogonal range queries. Chazelle [17] presented a minimal space structure for computing range queries in a $[1, f] \times [1, f]$ grid, that uses $f \log f(1 + o(1))$ bits and $O(f \log f)$ time to be built. It reports points in $O((1+occ') \log f)$ time, where occ' is the number of points reported. We want to use this data structure for the $[0, d'-1] \times [0, t-1]$ space, where d' is the number of nodes of $\mathcal{ST}$. However we only need to store f points. Therefore we must reduce the support spaces to rank spaces. The space $[0, d'-1]$ can be reduced to $[1, f]$ in $O(1)$ time, with $\text{R}\textsc{ank}$ over a bitmap of $d' + o(d')$ bits. The space $[0, t-1]$ requires more bits to be reduced. We store an array containing the $\text{D}\textsc{fs}$' values of the linking points. This array requires $(2 + \lceil \log t \rceil - \lfloor \log f \rfloor)f + o(f)$ extra bits using the representation of Grossi et al. [6]. The reduction is obtained in $O(\log f)$ time with a binary search over this array.

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.

3.2 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 $W(\textsc{Root}_{\mathcal{ST}}, p^R) : [0, t]$.

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 $2, 3, 4$, i.e. 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 don't need to consider words that start with b, since they don't 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 $cbd.bd.d.cba.ba.\underline{b}a$, $cbd.bd.d.cba.\underline{b}a.ba$, $cbd.bd.d.c\underline{b}a.ba.ba$ for ba and occurrences $cbd.\underline{b}d.d.cba.ba.ba$, $c\underline{b}d.bd.d.cba.ba.ba$, for bd.

Theorem 1. *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. $\square$

This algorithm was essentially presented by Navarro [4], except 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 theses 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 3. *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 $\textsc{Dfs}_{\mathcal{ST}}((\alpha.v)^R) \in W(\textsc{Root}_{\mathcal{ST}}, v^R)$. $\square$

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, i.e. 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. [1] for definitions). This procedure runs in $O(d)$ time. Note that the resulting structure remains a suffix tree.

3.3 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. [16] and related with the approach proposed by Ferragina et al. [3].

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*.

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 determined using the following dynamic programming equations:

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

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

We use Algorithm 2 to locate points $R(H(P[i..]))$ in $\mathcal{ST}$. Whenever it is not possible to descend by a letter, the $\text{DESCEND}_{\mathcal{ST}}$ procedure returns the $udef$ state. See table 2 (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. 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 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:

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

- 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 $\mathcal{ST}$.
- 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 $tail(P[i..])$.
- 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 2 (bottom). The only range query that finds occurrences (occ') is the $[5^*, 5^*] : [5, 8]$ query, as we have explained in this Section.

4 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 [18].

Definition 12. *The **LZ78 parsing** 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 Z_j is the largest prefix of $Z_i \ldots Z_n$ among the $Z_1, \ldots, Z_{i-1}$.*

Given a string T, we proceed as follows: compute the LZ78 parsing of $T^R = Z_1 \ldots Z_n$, then consider the suffix tree for strings $\{Z_1^R, \ldots, Z_n^R\}$ 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 4. *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$ 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. $\square$

Lemma 5. *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. $\square$

4.1 Space and Time Complexity

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.

Theorem 2. *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 summarised as follows:*

Space in bits	$[\frac{d}{n}(\frac{\lceil \log t\rceil - \lfloor \log d\rfloor}{\log u} + 1 + \epsilon) + \frac{d'}{n}(1+\epsilon) + \frac{f}{n}(\frac{\lceil \log t\rceil - \lfloor \log f\rfloor}{\log u} + 1)]uH_k$ $+o(u \log \sigma)$
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* $3u$ *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. Ziv et al. [18] showed that $\sqrt{u} \leq n \leq u/\log_\sigma u$, and, therefore $n = o(u \log \sigma)$. The relation between n and H_k was established by Kosaraju et al. [19] who showed that $n \log u = uH_k + o(u \log \sigma)$ for $k = o(\log_\sigma u)$.

(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 3 we know that the queries at the leaves must return occurrences. Therefore the total time amortises 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 don't 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 $\textsc{FirstLeaf}_{\mathcal{ST}}$. Therefore $R(\mathcal{T}_{78}(T))[i] = \textsc{Letter}_{\mathcal{ST}}(\textsc{FirstLeaf}_{\mathcal{ST}}, i)$. Hence we can compute the j-th letter of $R(\mathcal{T}_{78}(T))[i]$ in as $\textsc{Letter}(\textsc{Letter}_{\mathcal{ST}}(\textsc{FirstLeaf}_{\mathcal{ST}}, i), j)$, 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 [20], similar to Arroyuelo et al. [9]. 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 $\log u$ bits do appear as the first bits of some v. By $(i)_2$ we denote the binary representation of i, with $\log u$ 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 $2^{\log u} = u$ bits and can therefore be stored in $u + o(u)$ bits. With these bitmaps we are able to retrieve $\log u$ bits from a block in $O(1)$ time, i.e. $\log_\sigma u$ letters. We repeat these bitmaps for $\mathcal{ST}_{78}$ and hence are able to retrieve $\log u$ 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 another $u + o(u)$ bits. As pointed out by Arroyuelo et al. [9], 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 $(6.5+4\epsilon)H_k + o(u \log \sigma)$. However the worst example we were able to find, based on De Bruijn cycles, yielded $(5.5 + 3\epsilon)H_k + o(u \log \sigma)$ bits. In the next section we show concrete values for the space expression.

5 Practical Issues and Testing

We implemented a prototype for testing these ideas. It was pointed out by Navarro [4] that the range data structure was space consuming and actually slower in practice than to do a complete scan choosing the range that required less work. Therefore we did not implement the range data structure. Observe that this way we have no worst case guarantees for the search time.

The sparse suffix tree $\mathcal{ST}_{78}$ is stored in a suffix array fashion. The nodes of the $\mathcal{T}_{78}^R$ are stored as ranges over $\mathcal{ST}_{78}$, that correspond to the elements of $\mathcal{ST}_{78}$ that are traversed by the type 1 searches (see figure 1 for the range of node $2'$). 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 6 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 be able to read of edge-labels, by using suffix links. Every node stores its DFS time, a suffix link, the string depth, the HEAD pointer and the range of its corresponding R node.

We compared our implementation, Inverted-Lempel-Ziv-Index (ILZI), against Navarro's implementation of the FM-index (FMI), Sadakane's CSArray (CSAx1,CSAx8) and Navarro's LZ-Index (LZI), all of which are publicly available [21], using the files from the Pizza&Chili corpus [22][2].

We show the size of different indexes along with experimental values for the terms of the theoretical space requirements of our index, table 3. The FM-Index and the compressed suffix array needed to be parametrised. The parameters we used are also shown in table 3 in the *par* line. The parameter of the FM-Index was chosen with minimum value of 5 so that its size is close to the size of ILZI. The parameter of CSAx1 (resp. CSAx8) was chosen so that its size is close to the size of ILZI with $L = D$ (resp. $L = 8 \times D$). We used all the indexes to determine *occ* and reported this time divided by m as the counting time per character (c). We used all indexes to report occurrences, subtracted the counting time and divided by the number of occurrences found. We report this time as the reporting time per occurrence (r). Finally we used the indexes to display part of the text around the occurrences, subtracted the counting and reporting times, divided by the number of occurrences and letters. We report this time as the displaying time per character (o), also in table 3. The reporting time per occurrence is shown for different values of m, since for the LZ-based indexes this value is not constant. The time per occurrence and displaying time per character are relatively constant for different values of m and therefore we only present their values for $m = 20$.

In the space column of table 3 we present the ratios of the space size in bits with $u8$ and uH_k. In this, way for the raw string, we obtain the numbers of letters that should fit into a *byte*. Observe that our index has acceptable space requirements both in theory and in practice. For example for the xml file the practical value is $2.65uH_k$ bits and the theoretical value is $(2.62 + 1.62\epsilon)uH_k + o(u \log \sigma)$ bits.

The counting time per character of LZ-based indexes is affected by *occ*, whereas the FM-index and CSArray have a fairly constant value. This can be seen by the fact that the counting time per character decreases for larger values of m, where *occ* is smaller. By looking at the c lines of table 3 it can be seen that our reduction of 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 for counting when m is large. Also, for a large m, our index sometimes qualifies

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

File		Space						Time					
english		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	54.3	81.1	66.8	55.6	56.3	c 5	1.77e-3	6.78e-4	**1.30e-6**	3.41e-6	3.85e-6
	$i/u8$	1.00	1.09	1.62	1.34	1.11	1.13	c 10	4.33e-5	4.08e-5	**1.36e-6**	3.30e-6	3.80e-6
	i/uH_k	2.76	2.99	4.47	3.69	3.07	3.11	c 20	3.35e-6	3.01e-5	**1.19e-6**	2.92e-6	3.48e-6
	par				5	17	7	c 40	1.98e-6	3.17e-5	**1.06e-6**	2.43e-6	3.18e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	3.32e-7	**1.48e-7**	3.21e-5	7.28e-6	3.23e-6
	0.64	1.33	0.94	0.08	0.04	2.99 + 1.96 ϵ		o 20	3.09e-7	2.85e-7	**2.04e-7**	1.28e-6	9.16e-7
xml		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	26.1	44.5	64.9	26.2	25.8	c 5	4.37e-4	5.11e-4	**1.23e-6**	1.90e-5	5.02e-6
	$i/u8$	1.00	0.52	0.89	1.30	0.52	0.52	c 10	1.45e-4	1.69e-4	**1.30e-6**	1.41e-5	4.93e-6
	i/uH_k	5.08	2.65	4.52	6.60	2.67	2.62	c 20	3.25e-5	4.49e-5	**1.31e-6**	1.14e-5	4.86e-6
	par				5	44	19	c 40	6.18e-6	2.84e-5	**1.23e-6**	8.64e-6	4.68e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.40e-7**	4.67e-7	3.25e-5	2.00e-5	8.35e-6
	0.54	1.08	0.87	0.12	0.08	2.62 + 1.62 ϵ		o 20	2.84e-7	2.05e-7	**1.24e-7**	2.99e-6	1.97e-6
dna		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	44.0	60.9	63.4	45.1	37.0	c 5	1.93e-2	7.44e-3	**1.17e-6**	2.85e-6	4.72e-6
	$i/u8$	1.00	0.88	1.22	1.27	0.90	0.74	c 10	4.44e-4	1.76e-4	**1.42e-6**	3.57e-6	5.32e-6
	i/uH_k	3.63	3.19	4.42	4.60	3.27	2.69	c 20	3.51e-6	1.09e-5	**1.26e-6**	3.46e-6	5.16e-6
	par				5	26	11	c 40	1.66e-6	1.14e-5	**1.10e-6**	2.94e-6	4.92e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.61e-7**	3.98e-7	3.76e-5	1.37e-5	1.05e-5
	0.92	1.20	0.97	0.08	0.04	3.20 + 2.12 ϵ		o 20	2.93e-7	**2.62e-7**	7.78e-7	2.44e-6	2.66e-6
proteins		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	63.7	102.8	152.9	100.9	104.8	100.1	c 5	4.71e-4	1.88e-4	**1.27e-6**	3.12e-6	3.43e-6
	$i/u8$	1.00	1.61	2.40	1.58	1.64	1.57	c 10	3.77e-6	1.92e-5	**1.15e-6**	2.98e-6	3.37e-6
	i/uH_k	1.88	3.04	4.52	2.98	3.10	2.96	c 20	2.43e-6	2.16e-5	**1.03e-6**	2.51e-6	3.10e-6
	par				10	13	6	c 40	1.80e-6	2.30e-5	**9.53e-7**	1.88e-6	2.80e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**4.15e-7**	6.00e-7	1.69e-5	8.20e-6	6.47e-6
	0.85	1.22	0.98	0.04	0.04	3.11 + 2.07 ϵ		o 20	**3.12e-7**	4.27e-7	5.86e-7	1.11e-6	1.16e-6
pitches		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	53.2	84.7	124.8	86.8	85.6	86.1	c 5	2.58e-4	1.19e-4	**1.47e-6**	2.87e-6	3.06e-6
	$i/u8$	1.00	1.59	2.34	1.63	1.61	1.62	c 10	2.78e-5	3.78e-5	**1.34e-6**	2.68e-6	2.94e-6
	i/uH_k	1.99	3.16	4.66	3.24	3.19	3.21	c 20	1.15e-5	3.34e-5	**1.18e-6**	2.21e-6	2.60e-6
	par				9	12	5	c 40	6.78e-6	3.23e-5	**1.05e-6**	1.60e-6	2.21e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.39e-7**	4.85e-7	1.66e-5	5.60e-6	2.22e-6
	0.76	1.25	0.94	0.08	0.08	3.08 + 2.01 ϵ		o 20	2.66e-7	**1.45e-7**	6.33e-7	5.30e-7	4.06e-7
sources		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	53.5	80.9	68.1	53.3	54.6	c 5	8.91e-4	3.66e-4	**1.40e-6**	3.16e-6	3.48e-6
	$i/u8$	1.00	1.07	1.62	1.36	1.07	1.09	c 10	1.22e-4	6.43e-5	**1.42e-6**	3.00e-6	3.45e-6
	i/uH_k	2.80	3.00	4.53	3.81	2.99	3.06	c 20	1.58e-5	3.19e-5	**1.27e-6**	2.68e-6	3.21e-6
	par				5	17	7	c 40	3.60e-6	2.95e-5	**1.13e-6**	2.30e-6	2.89e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.33e-7**	4.56e-7	3.67e-5	7.17e-6	3.01e-6
	0.60	1.19	0.90	0.08	0.04	2.78 + 1.79 ϵ		o 20	2.83e-7	2.70e-7	**2.36e-7**	1.15e-6	7.92e-7

Table 3. Results for test files. On the left we show the space values and on the right the time values in seconds (s). In the space column we show the space requirements of different indexes, the original string (Raw), the Inverted-LZ-Index (ILZI), Navarro's LZ-index (LZI), Navarro's implementation of the FM-index (FMI) and Sadakane's CSArray (CSAx1,CSAx8). Variable i represents the size of the different indexes in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/u8$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/n$. The *par* line gives the parameters used for indexes that require it. For the CSArray we give the D value, for CSAx1 we have that $L = D$ and for CSAx8 that $L = 8 \times D$. The bottom part of the space column shows empirical values for the space terms of our index, d/n, d'/n, $((\lceil \log t \rceil - \lfloor \log d \rfloor)/\log u)$ in column DFS', $((\lceil \log t \rceil - \lfloor \log f \rfloor)/\log u)$ in column $\mathcal{L}$, (f/n) and in total the empirical value of the space expression. In the time column we show the time results for several query types. Lines labelled by c contains the counting cost per character, lines labelled by r the reporting time per occurrence, lines labelled by o the displaying time per character. The column labelled by m indicates the size of the pattern string used in the queries. The best values among different indexes are displayed in bold and the second best are underlined.

second, being faster than the CSArray. For $m = 40$ it is very close to the best counting time, expect for the xml and the pitches file where it is respectively around 5 times and 6.5 times slower than the FM-Index. Contrarily, 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 FM-Index and the CSArray.

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. This is also visible in table 3 as our index and LZI rank first and second and are one to two orders of magnitude faster than the alternatives.

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.

6 Conclusions

This paper presents two fundamental observations on LZ78 based compressed indexes. The first one is that our dictionary $\mathcal{T}_{78}$ is a suffix tree. This structure was first presented by Kärkkäinen [5] 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 [4] 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 [3] it appears as an FM-Index of $T^R_\$$. They present an argument to prove that its space requirements can be related to the entropy of the text T. However its suffix tree structure is also not explored. 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 behaviour. 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. At the same time, the structure we propose is smaller than LZI, for all the files we tested. In theory, with the terms we obtained in table 3, we can choose an ϵ to make the index smaller than $4uH_k + o(u \log \sigma)$. In practice it can be seen in table 3 that ILZI is always smaller than LZI. However a new version of the LZ-index proposed by Arroyuelo et al. [9] 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 1). This representation is not very competitive when compared to the compressed suffix trees presented by Sadakane [23]. 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. [9] 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.

Acknowledgements

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

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Second edn. McGraw (2001)
2. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms **48**(2) (2003) 294–313
3. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4) (2005) 552–581
4. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms **2**(1) (2004) 87–114
5. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proceedings of the 3rd South American Workshop on String Processing, Carleton University Press (1996) 141–155
6. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2) (2005) 378–407
7. Manzini, G.: An analysis of the burrows-wheeler transform. J. ACM **48**(3) (2001) 407–430
8. Makinen, V., Navarro, G.: Compressed full text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile (2006) 2nd version.
9. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proceedings of CPM 2006. LNCS 4009 (2006) 319–330
10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: Proceedings of SPIRE 2004. LNCS 3246, Springer (2004) 150–160 Extended version to appear in ACM TALG.
11. Grabowski, S., Mäkinen, V., Navarro, G.: First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In: Proceedings of SPIRE 2004. LNCS 3246, Springer (2004) 210–211
12. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1999)
13. Munro, J.I.: Tables. In Chandru, V., Vinay, V., eds.: Proceedings of FSTTCS 1996. Volume 1180 of LNCS., Springer (1996) 37–42
14. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. In: SODA, SIAM (2004) 1–10
15. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: ICALP. Volume 2719 of LNCS., Springer (2003) 345–356
16. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: COCOON. Volume 1090 of LNCS., Springer (1996) 219–230
17. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. **17**(3) (1988) 427–462
18. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24**(5) (1978) 530–536
19. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with lempel-ziv algorithms. SIAM J. Comput. **29**(3) (1999) 893–911
20. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: SODA, ACM Press (2006) 1230–1239
21. (http://www.dcc.uchile.cl/˜gnavarro/eindex.html)
22. (http://pizzachili.dcc.uchile.cl/)
23. Sadakane, K.: Compressed suffix trees with full functionality. (to appear in Theory of Computing Systems)