A Compressed Self-Index using a Ziv-Lempel Dictionary

Luís M. S. Russo\textsuperscript{1} \hspace{1cm} Arlindo L. Oliveira

March 22, 2006

\textsuperscript{1}Supported by the Portuguese Science and Technology Foundation by grant SFRH/BD/12101/2003 in project POCI 2010 and Project BIOGRID POSI/SRI/4778/2002
Abstract

This work presents a new compressed full-text self-index [11]. For a text with \( u \) characters compressible to \( n \) symbols by the LZ78 or LZW algorithm, we propose a compressed self-index that occupies \( O(n) \) space and reports the \( R \) occurrences of a pattern \( P \) of size \( m \) in \( O((m + R) \log n) \) time. To achieve this result we expose and explore the nature of a recurrent structure in LZ-indexes, the \( T_{78} \) suffix tree and point out the main obstacle to linear time algorithms based on LZ78 data compression. We show that our method is very competitive in practice by comparing it against the LZ-Index [13], the FM-index [5] and a compressed suffix array [14].
# Contents

1 Overview 2

2 Basic Concepts and Notation 3
   2.1 Strings .................................................. 3
   2.2 Suffix Trees ............................................ 3
   2.3 Suffix-links and Descend and Suffix Walks ............. 4

3 A Full-Text Index Using Suffix Tree Dictionaries 7
   3.1 Dictionaries ........................................... 7
   3.2 Occurrences Lying Inside a Single Block ............... 7
   3.3 Occurrences Spanning more than a Single Block ....... 8

4 A Compressed Self-Index based on LZ78 dictionaries 10
   4.1 General Overview and Space and Time Complexity .... 11

5 Practical Issues and Testing 12

6 Conclusions 14
Chapter 1

Overview

Compressed indexing is a recent and extremely successful line of research in text indexing. The idea is to explore text regularities to reduce the index space requirements much like what is done in data compression. A self-index can furthermore replace the original text string. Direct exploration of suffix arrays and the \( \psi \) function lead to the compressed suffix array approaches, for example Sadakane’s [14] compressed suffix array. Compression based on the Burrows-Wheeler technique lead to the FM-index approach [5]. Finally compression based on the Ziv-Lempel compression lead to LZ-indexes [13, 6, 9]. In this paper we make a contribution to the latter approach. Makinen and Navarro presented a comprehensive survey of compressed full-text indexes [11].

We start our exposure 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 self-index. Next we expose some of the practical decisions that were made to implement our algorithm. Finally we show some results and conclusions.
Chapter 2

Basic Concepts and Notation

2.1 Strings

Definition 1 A string $S$ is a finite sequence of symbols taken from a finite alphabet $\Sigma$ of size $\sigma$.

By $S[i]$ we denote the symbol at position $(i \mod |S|)$ of $S$. The empty string is denoted by $\epsilon$. The concatenation of two strings is denoted by $S.S'$. The size of a string $S$ is denoted by $|S|$. The set of strings of $\Sigma$ is denoted by $\Sigma^*$.

Definition 2 A point $S(i)$ in a string $S$ is the space in between letters $S[i – 1]$ and $S[i]$.

Definition 3 A prefix $S[.i – 1]$, substring $S[i..j]$, suffix $S[j+1..]$ of a string $S$ are (possible empty) strings such that $S = S[.i – 1].S[i..j].S[j+1..]$.

Definition 4 The reverse string $S^R$ of a string $S$ is the string such that $S^R[i] = S[.i – 1]$.

Definition 5 The exact matching problem consists in finding all occurrences of a (shorter) pattern string $P$ in a (longer) text string $T$, i.e. $O = \{i \mid T[i..i + |P| – 1] = P\}$. We denote $|P|$ by $m$, $|T|$ by $u$ and $\#O$ by $R$.

As a running example we shall consider string $T = cbdbddcbababa$. We have that $u = 13$, $\Sigma = \{a, b, c, d\}$, $\sigma = 4$, $T[0] = c$, $T[.1] = a$ and $T^R = abababcddbd$. Points $T(7)$ and $T(10)$ are shown below. The strings $cbdbdd = T[.6]$, $bab = T[7..9]$ and $aba = T[10..]$ are respectively a prefix, a substring and a suffix of $T$.

1
0123456 789 012
T: cbdbddcbababa
P: cbdbddcbababa

2.2 Suffix Trees

Suffix trees are optimal full-text indexes that can be stored in $O(u)$ space and built in $O(u)$ time, algorithms given by Weiner [16], McCreight [12] and Ukkonen [15].
Definition 6 A compact tree is a tree that has no nodes with unary child nodes.

Definition 7 A labelled tree is a tree that has a nonempty string label for every edge.

Definition 8 For a deterministic labelled tree the common prefix of any two edges out of a node is $\epsilon$.

Definition 9 A point $p$ in a labelled tree is either a node or a point in some edge-label.

By $DFS(p)$ we refer to the depth-first time of a point $p$ in a labelled tree considering that every point counts.

Definition 10 The path-label of a point $p$ in a labelled tree is the concatenation of the edge-labels from the root down to $p$. For deterministic trees it will also be denoted by $p$.

Definition 11 The string-depth of a point $p$ in a labelled tree is $|p|$.

Definition 12 The generalised suffix tree $T_{S_1,\ldots,S_k}$ of a set of strings $\{S_1,\ldots,S_k\}$ is the deterministic compact labelled tree such that the path-labels of the leaves are the suffixes of the $S_1\$, \ldots, $S_k\$ strings, where $\$ is a terminator symbol that does not belong to $\Sigma$.

We will refer to generalised suffix trees just as suffix trees. Whenever it is convenient we will omit the terminator symbol.

Definition 13 The range $I(p)$ of a point $p$ in a suffix tree $T$ is the interval of the DFS values of the points that are descendents of $p$.

Definition 14 The reverse tree $Tr^R$ of a labelled tree $Tr$ is the minimal deterministic labelled tree that for every node $v$ of $Tr$ contains a node $v^R$.

Therefore we can define a canonical mapping $R$ that for every node $v$ in $Tr$ maps $DFS(v)$ to $DFS(v^R)$.

As our running example, consider $T$ as the suffix tree in figure 2.1 (top-right). We have that $DFS(c) = 5$, $DFS(db) = 6$, $I(c) = [5, 8]$. $T^R$ is also in figure 2.1 (top-left). Observe that the reverse tree of a suffix tree is always a trie.

2.3 Suffix-links and Descend and Suffix Walks

Definition 15 The suffix-link of a node $v$ of a suffix tree is a pointer to node $v[1..]$. Typical algorithms over a suffix tree $T$ consist 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 16 The descend and suffix walk of a string $P$ over a suffix tree $T$ is the sequence $p_0 \ldots p_{|P|}$ of points of $T$ computed by algorithm 1.

Definition 17 The right, left traces of a string $P$ over a suffix tree $T$ are the subsequences of the descend and suffix walk given respectively by lines 6 and 8 of algorithm 1.
Figure 2.1: (top-right) Suffix tree for strings \{a, b, ba, bd, cba, cb, d\}. Suffix link from \(cb\) to \(b\) shown by a dashed arrow. Nodes show their DFS value. (top-left) Reverse tree of the suffix tree on the right. Nodes show the DFS value. The \(R\) mapping is shown and \(R(3)\) is indicated by a bold arrow. (bottom-left) Sparse suffix tree of \(T\). (bottom-right) 2-D range data and \([3*,3*];[5,8]\) query.

Note that we define in an artificial way \texttt{SUFFIX\_LINK(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 new terminator character \(\$\) that fails to match with any other character. For the theoretical part of this paper we assume that \texttt{DESCEND} is implemented using perfect hash functions and can hence be computed in \(O(1)\) time\(^1\), i.e. it is alphabet independent. A descend and suffix walk can be computed in \(O(m)\) amortised time \([8]\). Table 2.2 shows the descend and suffix walk of \(cbd\text{bd}d\text{dc}\).

\(^1\)Suggested by Gusfield \([8]\).
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[i]</td>
<td>c</td>
<td>b</td>
<td>d</td>
<td>b</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>trace-left[i]</td>
<td>ε</td>
<td>c</td>
<td>cb</td>
<td>cbd</td>
<td>b</td>
<td>bd</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>DFS(father)</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>DFS(trace-left[i])</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>DFS(child)</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>trace-right[i]</td>
<td>cbd</td>
<td>bd</td>
<td>d</td>
<td>bd</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>ε</td>
</tr>
<tr>
<td>DFS(father)</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DFS(trace-right[i])</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>I(trace-right[i])</td>
<td>[8,8]</td>
<td>[4,4]</td>
<td>[9,9]</td>
<td>[4,4]</td>
<td>[9,9]</td>
<td>[9,9]</td>
<td>[5,8]</td>
<td>[0,9]</td>
</tr>
<tr>
<td>DFS(child)</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>P[i..]</td>
<td>cbd.b.d.c</td>
<td>bd.b.d.c</td>
<td>d.b.d.c</td>
<td>bd.d.c</td>
<td>d.d.c</td>
<td>d.c</td>
<td>c</td>
<td>ε</td>
</tr>
<tr>
<td>tail(P[i..])</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>ε</td>
</tr>
<tr>
<td>H(P[i..])</td>
<td>849</td>
<td>449</td>
<td>949</td>
<td>49</td>
<td>99</td>
<td>9</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>R(H(P[i..]))</td>
<td>6'7'8'</td>
<td>6'7'7'</td>
<td>6'7'6'</td>
<td>6'7'</td>
<td>6'6'</td>
<td>6'</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>Min(P,i)</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
<td>6'7'8'</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
</tr>
<tr>
<td>Max(P,i)</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
<td>6'7'8'</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
<td>udef</td>
</tr>
<tr>
<td>L(P,i)</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
<td>[3*..3*]</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>I(tail(P[i..]))</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[5,8]</td>
<td>[0,9]</td>
</tr>
<tr>
<td>occ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: (Top) Descend and suffix walk of cbdbdcd in T. (Bottom) Values for locating type > 1 occurrences.

Algorithm 1 Descend and Suffix Walk Algorithm

1: procedure Descend_and_Suffix(P)
2: P ← PS
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 ← SUFFIX_LINK(point)
11: end while
12:  point ← DESCEND(point, P[i])
13: end for
14: end procedure
Chapter 3

A Full-Text Index Using Suffix Tree Dictionaries

3.1 Dictionaries

Throughout section 3 we assume we are given an arbitrary suffix tree $T$ and we will use it as a dictionary. We consider as dictionary words the path-labels of the nodes of $T$.

**Definition 18** The translation $V(v_1 \ldots v_f)$ of a sequence $v_1 \ldots v_f$ of nodes is a string such that $V(v_1 \ldots v_f)[i] = DFS(v_i)$.

We also extend the canonical mapping $R$ to sequences in the following way $R(v_1 \ldots v_f) = R(v_f) \ldots R(v_1)$.

**Definition 19** The $T$-maximal parsing of a string $T$ is the sequence $v_1, \ldots, v_f$ of nodes of $T$ 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 $T$.

**Definition 20** The head, tail of the $T$-maximal parsing of $T$ 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 $T$.

We denote by $T(T)$ the translation of the $T$-maximal parsing of $T$ and by $H(T)$ the translation of its head. In our example the $T$-maximal parsing of a string $T$ is the sequence $cbd, bd, d, cha, ba, ba$, the head is the sequence $cbd, bd, d, cha, ba$ and the tail is string $ba$. Hence $T(T) = 849733$ and $H(T) = 84973$. Finally $R(849733) = R(3)R(3)R(7)R(9)R(4)R(8) = 223678$.

3.2 Occurrences Lying Inside a Single Block

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. To find occurrences of type 1 we require $T$, the reverse tree $T^R$ and $T(T)$. The approach for finding occurrences of $P$ inside a block was presented by Navarro [13] and is the following:
Degrad by P in T. If this is impossible then there are no type 1 occurrences of P.

- Start a search in T from point P.

- For each node p reached in the previous step, start a search in $T^R$ from the node $p^R$.

- Report every block that corresponds to a node reached in $T^R$.

**Theorem 1** The above procedure is correct and complete.

**Proof 1** (Correct) Clearly every reported block is of the form $\alpha.P.\beta$ for some $\alpha, \beta$ and hence it contains an occurrence of P.

(Complete) Suppose block $v_1 = \alpha.P.\beta$, hence $\alpha.P.\beta$ is a node in T. Since T is a suffix tree $P.\beta$ is also a node in T. Node $P.\beta$ is reached by the search in T since it starts by P. Node $(\alpha.P.\beta)^R$ is reached by the search in $T^R$ since it starts by $(P.\beta)^R$, hence block $v_1$ is reported.

Consider $P = b$. This pattern has only type 1 occurrences. By reading b we reach node 2 of T (see fig. 2.1). The search on T returns nodes 2, 3, 4. The following nodes of tree $T^R$ will be visited, $R(2), R(6)$ because of 2, $R(3), R(7)$ because of 3 and nodes $R(4), R(8)$ because of 4. Hence we found 5 matches of pattern b in T.

We can store additional pointers in the T and $T^R$ trees in order to avoid nodes with no occurrences. This way the above algorithm runs in $O(m + R_1)$ time, where $R_1$ is the number of type 1 pattern occurrences.

3.3 Occurrences Spanning more than a Single Block

This section is largely based on ideas presented by Karuskainen and Ukkonen [10]. It is related with the approach proposed by Ferragina and Manzini [6].

**Definition 21** The sparse suffix tree $ST$ of a string T and a suffix tree T is the suffix tree of $R(T(T))$.

The sparse suffix tree of our example is in figure 2.1 (bottom-left). Our algorithm starts by locating strings $R(H(P[i...]))$ in ST. According to the definition of T-maximal parsing the best way to achieve this is by using dynamic programming. The idea is to extend the $H(P[i...])$ from right to left. The only thing we must determine is $H(P[i...])[0]$ and this can easily be obtained from the right trace of descend and suffix walk of P in T. At each step we descend in ST from $R(H(P[i...])[1...])$ by $R(H(P[i...])[0])$. This process takes $O(m)$ time and space (see table 2.2 for an example).

If extending $H(P[i...])$ to the left overlaps the whole pattern we must consider all the $v_j$'s that have $P[i...i-1]$ as a suffix, i.e. the descendents of $R(P[i...i-1])$ in $T^R$.

**Definition 22** Consider the set of points in ST:

$$E_i = \{ R(J, H(P[i...])) \mid R(J) \text{ is a descendant of } R(P[i...i-1]) \text{ in } T^R \}$$

By minimal extension $\text{Min}(P, i)$ we refer to the lexicographically smallest element of $E_i$.

By maximal extension $\text{Max}(P, i)$ we refer to the lexicographically largest element of $E_i$.

---

1 This is not the original definition but it is similar.
Locating $Min(P,i)$ and $Max(P,i)$ in $ST$ after computing the $R(H(P[i..]))$ points is achieved by doing binary searches in the array of child nodes. This takes $O(\log d)$ where $d$ is the size of $T$. As an example, consider $P = c0dabcd$. We have that $H(P[3..]) = 49$. Hence $R(H(P[3..])) = 6'7'$ and $E_3 = \{6'7'8'\}$. Therefore, both the $Min(P,i)$ and $Max(P,i)$ are equal to $6'7'8'$.

We assume that the leaves of $ST$ are lexicographically ordered. We denote this value by $LEX$.

**Definition 23** The leaf range $L(P,i)$ is the interval of $LEX$ values of leaves that are descendents of some element of $E_i$.

Clearly we can compute this range by knowing $Min(P,i)$ and $Max(P,i)$. In our example $L(P,3) = [3^*, 3^*]$ and $LEX(6'7'8'8') = 3^*$.

Finally we need a data structure that relates the leaf ranges and the tails from the $T$-maximal parsing of $P[i..]$. We will use a two-dimensional range data structure. Let $v_1 \ldots v_f$ be the $T$-maximal parsing of $T$. Then for every $0 < i < f$ we store the point $(LEX(R(V(v_1 \ldots v_i))$), $DFS(p_i))$, where $p_i$ is the largest prefix of $v_{i+1} \ldots v_f$ that is point in $T$.

In our example this gives the following points:

- $(LEX(R(84973))$, $DFS(ba)) = (LEX(2'3'6'7'8'8'), 3) = (1^*, 3)$
- $(LEX(R(8497))$, $DFS(ba)) = (LEX(3'6'7'8'8'), 3) = (2^*, 3)$
- $(LEX(R(849))$, $DFS(cba)) = (LEX(6'7'8'8'), 3) = (3^*, 7)$
- $(LEX(R(84))$, $DFS(d)) = (LEX(7'8'8'), 3) = (4^*, 9)$
- $(LEX(R(8))$, $DFS(bd)) = (LEX(8'8'), 3) = (5^*, 4)$

Figure 2.1 (bottom-right) shows our set of points. Our algorithm for finding type $1 > 1$ occurrences of $P$ proceeds as follows:

- Compute the descend and suffix walk of $P$ in $T$.
- Locate the $R(H(P[i..]))$ points in $ST$.
- Compute $Min(P,i)$ and $Max(P,i)$, i.e. obtain $L(P,i)$.
- Obtain $I(tail(P[i..]))$ from the right trace of the descend and suffix walk.
- Compute the orthogonal range queries for $L(P,i)$ and $I(tail(P[i..]))$.

An example of our algorithm is shown in table 2.2 (bottom). The only range query that finds occurrences (occ) is the $[3^*, 3^*] : [5, 8]$ query, see figure 2.1.

To answer orthogonal range queries, we use a minimal space structure by Chazelle [4]. It requires $O(f)$ computer words, $O(f \log f)$ time to build and answers queries in $O((1 + occ) \log f)$ time, where $occ$ is the number of occurrences reported.
Chapter 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 [17].

**Definition 24** 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 $T_{78}$. In our example $T^R$ is parsed into $a, b, ab, abc, d, db, dbc$ and the resulting dictionary can be seen in figure 2.1 (top-right).

The following lemmas expose why the dictionary we propose is adequate in terms of space.

**Lemma 1** If the number of blocks of the LZ78 parsing of $T$ is $n$ then the $T_{78}$ suffix tree can be stored in $O(n)$ space.

**Proof 2** First we argue that any suffix tree with $n$ leaves can be stored in $O(n)$ space. Basically the problem is that we can’t store the edge labels because they might require $O(n^2)$ space\(^1\).

The classical solution for this problem consists in storing the string $Z_1^R \ldots Z_n^R$ and pointers to it. Clearly this is unsuitable since it would require $O(n)$ space in our case. In every node $v$ we store the letter $v[0]$ and if its father node is $v[\ldots i-1]$ we store a head pointer to node $v[i..]$. This information suffices to allow reading of edge-labels sequentially by using suffix links. We lose the ability to access a random letter in an edge-label but this is not relevant for descend and suffix walks.

Now we determine the number of leaves of $T_{78}$, i.e. the size of the set $L = \{p | p \text{ is a leaf of } T_{78}\}$. Obviously $L \subseteq \{Z_1^R, \ldots, Z_n^R\}$. In fact since every suffix of a $Z_i^R$ is a $Z_j^R$ for some $j$, we have that $L = \{Z_1^R, \ldots, Z_n^R\}$. Hence $\#L = n$, i.e. $T_{78}$ has $n$ leaves and therefore requires $O(n)$ space.

**Lemma 2** If the number of blocks of the LZ78 parsing of $T$ is $n$ then the $T_{78}$-maximal parsing of $T$ has at most $n$ blocks.

\(^1\)Shit register sequences are a simple alphabet independent example.
**Proof 3** The idea is to show that if a block $v_i$ of the $T_{78}$-maximal 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 $Z_j^R$. □

### 4.1 General Overview and Space and Time Complexity

With the previous lemmas and since $n \log n = O(u \log \sigma)$ our index can be built in $O(u \log \sigma)$ time and $O(n)$ space by the following procedure:

- Compute the LZ78 parsing of $T^R$ (time: $O(u \log \sigma)$, space: $O(n)$).
- Build suffix tree $T_{78}$ and reverse tree $T_{78}^R$ (time: $O(u \log \sigma)$, space: $O(n)$).
- Compute the $T$-maximal parsing of $T$ (time: $O(u)$, space: $O(u)$).
- Build the sparse suffix tree $ST$ of $T$ (time: $O(n)$, space: $O(n)$).
- Build the range data structure and discard $T$ (time: $O(u \log \sigma)$, space: $O(n)$).

Our index answers queries in time $O((m + R_{>1}) \log n + R_1)$ using the following procedure:

- Compute the descend and suffix walk of $P$ in $T_{78}$ ($O(m)$).
- Find type 1 occurrences ($O(R_1)$).
- Find type > 1 occurrences ($O((m + R_{>1}) \log n)$).
Chapter 5

Practical Issues and Testing

It was pointed out by Navarro [13] 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.

Sparse suffix trees $ST$ are stored in a suffix array like manner. In fact it suffices to store an array ordered by the letter nearest to the root. The reverse tree $T^R_8$ is not stored. The idea is to replace every node in $T^R_8$ store by a range in $ST$ that contains the occurrences of its descending nodes (see the bold arrows in figure 2.1).

**Definition 25** A *spurious* entry for string $T$ in the suffix tree $T$ is a leaf $v$ of $T$ such that $v^R$ is a leaf of $T^R$ and $v$ is not a block in the $T$-maximal parsing of $T$.

A technique that produced rather good results was to remove spurious entries from dictionary after the maximal parsing.

We compared our implementation, inverted Lempel Ziv Index (ILZI), against Navarro’s implementation of the FM-index (NAV-FMI), Sadakane’s CSArray (CSA) and Navarro’s LZ-index (LZI), all of which are publicly available [1]. We used two files from the Pizza&Chili corpus [2]. The first, English, contains 200 megabytes (Mb) of English from the Gutenberg Project. The second, proteins, contains 64 megabytes (Mb) of proteins from the Swissprot database.

We tested on a Pentium 4 Dual-Core at 3.2 GHz, 1 Mb of cache and 1Gb of RAM running Fedora Core 3. The code was compiled with gcc 3.4.4 and optimised with -09. Times were obtained using 2^{15} repetitions.

We can observe, in figure 5.1, that our index performs better than all other indexes either in space or time for every pattern size when reporting or outputting occurrences in English.

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>LZI</th>
<th>NAV-FMI</th>
<th>CSA</th>
<th>ILZI</th>
</tr>
</thead>
<tbody>
<tr>
<td>english.200MB</td>
<td>200</td>
<td>316</td>
<td>268</td>
<td>208</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td></td>
<td>158 %</td>
<td>134 %</td>
<td>104 %</td>
<td>103 %</td>
</tr>
<tr>
<td>proteins</td>
<td>64</td>
<td>153</td>
<td>104</td>
<td>111</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>240 %</td>
<td>164 %</td>
<td>175 %</td>
<td>161 %</td>
</tr>
</tbody>
</table>

Table 5.1: Space usage of different compressed indexes in megabytes (Mb) and size ratio. Inverted LZ-Index (ILZI) Navarro’s implementation of the FM-index (NAV-FMI), Sadakane’s CSArray (CSA) and Navarro’s LZ-index (LZI).
Figure 5.1: Average user times for counting (top), reporting (middle) and outputting (bottom) occurrences of patterns of size $m$ in files English (left) and proteins (right).

or outputting occurrences in proteins. For counting queries our index achieves an acceptable performance for $m = 10$ in proteins and $m = 20$ in English.
Chapter 6

Conclusions

This paper presents two fundamental observations on LZ78 based compressed indexes. The first one is that our dictionary $T_{78}$ is a suffix tree and that we can store it in a practical and simple way in $O(n)$ space (see the proof of lemma 1). This structure was first presented by Kärkkäinen [9] 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 [13] 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 [7] it appears as an FM-Index of $T^R$. There an argument is presented to prove that its space requirements can be related to the entropy of the text $T$ but its suffix tree structure is also not explored. The second fundamental problem that one needs to solve in order to obtain a linear time algorithm is that $P$ may occur with $m^2$ different $H(P[i..])$ strings in the LZ78 parsing. We solve this problem by discarding the original parsing and using a maximal parsing. 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.

The main problem with the solution presented by Ferragina and Manzini is that it relies heavily on the FM-Index and hence may have alphabet related problems. Some solutions have been presented to address this problem [7]. Our approach is simpler and alphabet independent. Also, in practice, the $\log n$ branching factor is irrelevant since the sparse suffix tree branches heavily only at the root.

Experimental results show that our prototype is among the most competitive for reporting and outputting occurrences and also performs reasonably well for counting occurrences when the pattern is large.
Acknowledgements

We are grateful to Gonzalo Navarro and Sadakane for providing testing prototypes. We also thank Gonzalo Navarro and Paolo Ferragina for the Pizza&Chili Corpus. We thank Gonzalo Navarro for organizing a Workshop on Compression, Text, and Algorithms at DCC on November 8 that allowed for stimulating discussions. We also would like to thank Rodrigo Gonzalez, Diego Arroyuelo and Pedro Morales for discussions on compressed indexes.

Finally we would like to thank Luis Coelho for countless discussions about our index.
Bibliography


