PhD Thesis Proposal

Ziv-Lempel Compressed Full-Text Self-Indexes

PhD Student: Diego Arroyuelo Billiardi (`darroyue@dcc.uchile.cl`)
Advisor: Prof. Gonzalo Navarro (`gnavarro@dcc.uchile.cl`)

Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, Santiago, Chile.

Abstract. Given a sequence of characters $T_{1...u}$ (the text) over an alphabet Σ, and given another sequence $P_{1...m}$ (the *search pattern*) over Σ, then the *full-text search problem* consists of finding (or counting, or reporting) all the *occ* occurrences of P in T. In *indexed text searching* we build a data structure (or *index*) on the text to restrict the search to a small portion of the text, improving search time but increasing the space requirement to solve the problem. The current trend in indexed text searching is that of *compressed full-text self-indexes*, which *replace* the text with a more space-efficient representation of it, and at the same time this representation provides indexed access to the text.
In this thesis we propose a deep study of compressed full-text self-indexes based on the Ziv-Lempel compression algorithm, contributing with new theoretical developments in this area. Specifically, we will focus our studies on the Navarro's LZ-index [38–40]. We aim at a compressed full-text self-index with many interesting properties: fast full-text searching and text recovery; using little space for construction and operation; allowing insertion and deletion of text; providing a range of space/time trade-offs; and efficient construction and search in secondary memory.

1 Introduction and Previous Work

Text searching is a classical problem in Computer Science. Given a sequence of characters $T_{1...u}$ (the text) over an alphabet Σ of constant size σ, and given another (short) sequence $P_{1...m}$ (the *search pattern*) over Σ, then the *full-text search problem* consists of finding (or counting, or reporting) all the *occ* occurrences of P in T. There are two general approaches for solving the full-text searching problem:

Sequential Text Searching: we search for the pattern P directly on the plain representation of T. That is, we do not construct any data structure on the text, mainly because the text is small, highly dynamic, or it is not available in advance. See [42] for a complete review on sequential text searching.

Indexed Text Searching: we build a data structure (or *index*) on the text to restrict the search to a small portion of the text, improving search time but increasing the space requirement to solve the problem. This approach is used when the text is so large that a sequential scanning is prohibitively costly, many searches (using different patterns) must be performed on the same text, the text does not change so frequently, and there is sufficient storage space to maintain the index and provide efficient access to it. In this thesis we focus on indexed text searching.

A *full-text database* is a system providing fast access to a large mass of textual data. By far its most challenging requirement is that of performing fast text searching for user-entered patterns. Typical text databases contain natural language texts, DNA or protein sequences, MIDI pitch sequences, program code, etc. Modern text databases have to provide fast access to the text, using as little space as possible. These goals are opposed because, in order to provide fast access, an index has to be built on the text. This index is a data structure stored in the database, hence increasing the space requirement. In recent years there has been much research on *compressed text databases*, focusing on techniques to represent the text and the index using little space, yet permitting efficient text searching.

Text compression is a technique to represent a text using less space. It has two main advantages [3]:

– *Advantage 1:* It reduces the space requirement of the text, and
– *Advantage 2:* It reduces the cost and increases the effective speed of text transmission, both between computers in a network and from secondary to main memory (where a compressed text is read faster than its uncompressed form).

The main disadvantage of compression is that processing time is increased, but the accesses to secondary memory are reduced (that is, there is a trade-off CPU time/secondary memory accesses). A concept related to text compression is that of the k-th order empirical entropy of a text T, denoted by $H_k(T)$ [32]. The value $uH_k(T)$ provides a lower bound to the number of bits needed to compress T using any compressor that encodes each character considering only the context of k characters that precede it in T. Also it holds that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$ (log means $\log_2$ in this proposal).

Despite that there has been some work on space-efficient inverted indexes for natural language [47, 41] (able of finding whole words and phrases), until a short time ago it was believed that any general index for text searching (such as those that we are considering in this thesis) would need much more space. In practice, the smallest indexes available were the suffix arrays [31], requiring $u \log u$ bits to index a text of u characters. Since the text requires $u \log \sigma$ bits to be represented, this index is usually much larger than the text (typically 4 times the text size). With the huge texts available nowadays (for example, the human genome consists of about 3×10^9 base pairs), one solution is to store the indexes on secondary memory. However, this has significant influence on the running time of an application, as access to secondary memory is considerably slower.

Since the last decade, several attempts to reduce the space of the suffix trees [2] or arrays have been made by Kärkkäinen [19], Kurtz [25], Mäkinen [27], and Abouelhoda et al. [1]. These approaches have been mainly practical, in the sense that it was easy to obtain an implementation from the algorithmic formulation, and the results have been remarkable, but not spectacular.

A parallel, more principled track, started at about the same time, thanks to Kärkkäinen and Ukkonen [22, 23, 20], Grossi and Vitter [15], Sadakane [44, 45], Ferragina and Manzini [9–11], and later Grossi, Gupta and Vitter [13], Navarro [38–40], and Mäkinen and Navarro [28, 29]. All these works present *compressed indexes*, which take advantage of the regularities of the text to operate in space proportional to that of the compressed text (e.g., 3 times the zero-order entropy of the text). Especially, in some of those works [44, 45, 9–11, 13, 14, 38–40, 28, 29, 12], the indexes *replace* the text and, using little space (sometimes even less than the original text), provide indexed access. This feature is known as *self-indexing*, since the index allows one to search and retrieve any part of the text without storing the text itself. This is an unprecedented breakthrough in text indexing and compression.

As with text compression, using compressed indexes increases processing time. However, given the relation between main and secondary memory access times, it is preferable to handle compressed indexes entirely in main memory, rather than handling them in uncompressed form but in secondary storage. When the compressed index is so large that it does not fit in main memory, then *Advantage 2* allows to reduce the cost of transmission between secondary and main memory, since a smaller index potentially requires less disk accesses.

Most works on compressed full-text indexes are based on suffix arrays [44, 45, 9, 10, 13, 14, 28, 29, 12]. However, there exist other works based on Ziv-Lempel compression algorithm [48]. Below we review those Ziv-Lempel-based works. An important property is that, if the Ziv-Lempel parsing cuts the text into n *phrases* (see Section 2), then $n \log u = uH_k(T) + o(kn \log \sigma)$ for any k [24].

Kärkkäinen and Ukkonen [22, 20] propose a suffix tree that indexes only the beginnings of the blocks produced by a Ziv-Lempel compression. This index requires $O(\frac{1}{\epsilon} uH_k(T)) + O(u \log\log u / \log u) + u \log \sigma$ bits (the last term is for the text), and the occ occurrences of P in T can be found in $O(m^2 + mu^\epsilon + occ)$ and even in $O(m^2 + m \log u + \frac{1}{\epsilon} occ \log^\epsilon n)$ time, depending on the structures used [20], where $k = O(\log_\sigma \log u)$ and $0 < \epsilon < 1$ (not necessarily a constant). As can be seen, this is not a self-index since it needs the text to operate. On the other hand, Ferragina and Manzini [11] present an index based on Ziv-Lempel compression, although combined with Burrows-Wheeler compression [4]. This is the only existing compressed full-text self-index taking $O(m + occ)$ time to locate the occ occurrences of P in T. The index requires $O(uH_k(T) \log^\gamma u)$ bits of storage, for any constant $\gamma > 0$. Finally, Navarro [38–40] presents the LZ-index, which is a full-text self-index based on the Ziv-Lempel parsing of the text. If the text is parsed into n phrases by the LZ78 algorithm, then the LZ-index takes $4n \log n(1 + o(1))$ bits of space, which is 4 times the size of the compressed text and also 4 times the k-th order text entropy, i.e. $4uH_k(T) + o((1 + H_k(T))u)$, for any $k = O(\log_\sigma \log u)$ [24, 11]. The LZ-index answers queries in $O(m^3 \log \sigma + (m + occ) \log n)$ worst case time. The index also replaces the text (that is, is a *self-index*): it can reproduce a text context of length L around an occurrence found (and in fact any sequence of phrases) in $O(L \log \sigma)$ time, or obtain the whole text in time $O(u \log \sigma)$. The index is built in $O(u \log \sigma)$ time.

The most basic problems for compressed self-indexes are that of searching and reproducing the text. However, there are many other functionalities that a self-index must provide in order to be fully useful. Many of those have been obtained separately in the indexed text searching literature. For example, there are

– indexes like the suffix trees, allowing to search for a pattern in optimal $O(m+occ)$ time;
– the suffix arrays, which are one of the most used indexes in practice;
– compressed indexes using little space (and many times including the text);
– indexes requiring little space to be built [26, 16, 18]: Compressed indexes are usually derived from a classical one. Although it is usually simple to build a classical index and then derive its compressed version, there might not be enough space to build the classical index first. Secondary memory might be available, but many classical indexes are costly to build in secondary memory. Therefore, an important problem is how to build compressed indexes without building their classical versions first;
– indexes allowing efficient construction and search in secondary memory [8, 6]: Although their small space requirements might permit compressed indexes fit in main memory, there will always be cases where they have to operate on disk. There is not much work yet on this important issue. A good survey on full-text indexes in secondary memory is by Kärkkäinen and Rao [21];
– and others allowing efficient insertion and deletion of texts [8, 9, 17, 5]: Most indexes in the literature are static, in the sense that they have to be rebuilt from scratch upon text changes. This is currently a problem even on uncompressed full-text indexes, and not much has been done.

However, no existing data structure for text searching fits all the requirements.

In this thesis we propose a deep study of compressed full-text self-indexes based on the Ziv-Lempel compression algorithm. Specifically, we will focus our studies on Navarro's LZ-index. We aim at a compressed full-text self-index with the following properties:

– Fast full-text searching,
– Fast text recovery,
– Using little space for construction and operation,
– Allowing insertion and deletion of text,
– Providing a range of space/time trade-offs, and
– Efficient construction and search in secondary memory (using *Advantage 2* to improve performance).

Currently, the LZ-index has the following properties: fast full-text searching, fast text recovery, uses little space for operation, does not allow insertion nor deletion of text, only operates in main memory, and it needs much construction space.

2 Ziv-Lempel Compression

The general idea of Ziv-Lempel compression is to replace substrings in the text by a pointer to a previous occurrence of them. If the pointer takes less space than the string it is replacing, compression is obtained. Different variants over this type of compression exist, see for example [3]. We are particularly interested in the LZ78/LZW format, which we describe as follows.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [48]) is based on a dictionary of blocks (or *phrases*), in which we add every new block computed. At the beginning of the compression, the dictionary contains a single block b_0 of length 0. The current step of the compression is as follows: if we assume that a prefix $T_{1\ldots j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \ldots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1\ldots u}$ which is a block of the dictionary. Once we have found this block, say b_s of length ℓ_s, we construct a new block $b_{r+1} = (s, T_{j+\ell_s+1})$, write the pair at the end of the compressed file Z, i.e. $Z = b_1 \ldots b_r b_{r+1}$, and add the block to the dictionary. It is easy to see that this dictionary is prefix-closed (that is, any prefix of an element is also an element of the dictionary) and a natural way to represent it is a trie.

LZW [46] is just a coding variant of LZ78, so we will focus in LZ78 in this thesis, understanding that the algorithms can be trivially ported to LZW.

An interesting property of this compression format is that every block represents a different text substring. The only possible exception is the last block. We use this property in our algorithm, and deal with the exception by adding a special character "\$" (not in the alphabet and considered to be smaller than any other character) at the end of the text. The last block will contain this character and thus will be unique too.

The compression algorithm is $O(u)$ time in the worst case and efficient in practice if the dictionary is stored as a trie, which allows rapid searching of the new text prefix (for each character of T we move once in the trie).

Another concept that is worth reminding is that a set of strings can be lexicographically sorted, and we call the *rank* of a string its position in the lexicographically sorted set. Moreover, if the set is arranged in a trie data structure, then all the strings represented in a subtree form a lexicographical interval of the universe. We remind that, in lexicographic order, $\varepsilon \leqslant x$, $ax \leqslant by$ if $a < b$, and $ax \leqslant ay$ if $x \leqslant y$, for any strings x, y and characters a, b.

3 The LZ-index Data Structure

Suppose that the text $T_{1\ldots u}$ has been partitioned using the LZ78/LZW algorithm into $n+1$ blocks $T = B_0 \ldots B_n$, such that $B_0 = \varepsilon$; $\forall k \neq \ell$, $B_k \neq B_\ell$ (that is, no two blocks are equal); and $\forall k \geqslant 1$, $\exists \ell < k$, $c \in \Sigma$, $B_k = B_\ell \cdot c$ (that is, every block except B_0 is formed by a previous block plus a letter at the end).

3.1 Data Structures

The following data structures conform the LZ-index [38–40]:

1. *LZTrie*: is the trie formed by all the blocks $B_0 \dots B_n$. Given the properties of LZ78 compression, this trie has exactly $n+1$ nodes, each one corresponding to a string. *LZTrie* stores enough information so as to permit the following operations on every node x: (a) $id_t(x)$ gives the node identifier, i.e., the number k such that x represents B_k; (b) $leftrank_t(x)$ and $rightrank_t(x)$ give the minimum and maximum lexicographical position of the blocks represented by the nodes in the subtree rooted at x, among the set $B_0 \dots B_n$; (c) $parent_t(x)$ gives the tree position of the parent node of x; and (d) $child_t(x, c)$ gives the tree position of the child of node x by character c, or *null* if no such child exists. Additionally, the trie must implement the operation $rth_t(rank)$, which given a rank r yields the block identifier representing the lexicographically r-th string of $\{B_0, \dots, B_n\}$.
2. *RevTrie*: is the trie formed by all the reverse strings $B_0^r \dots B_n^r$. For this structure we do not have the nice properties that the LZ78/LZW algorithm gives to *LZTrie*: there could be internal nodes not representing any block. We call these nodes *empty*. We need the same operations for *RevTrie* than for *LZTrie*: id_r, $leftrank_r$, $rightrank_r$, $parent_r$, $child_r$, and rth_r.
3. *Node*: is a mapping from block identifiers to their node in *LZTrie*.
4. *Range*: is a data structure for two-dimensional searching in the space $[0 \dots n] \times [0 \dots n]$. The points stored in this structure are

$$\{(revrank(B_k^r), rank(B_{k+1})), k \in 0 \dots n-1\},$$

   where $revrank$ is the lexicographic rank in $\{B_0^r \dots B_n^r\}$ and $rank$ is the lexicographical rank in $\{B_0 \dots B_n\}$. For each such point, the corresponding k value is stored.

Each of these data structures requires $n \log n(1+o(1))$ bits of storage, which makes the index space $4uH_k(T)(1+o(1))$ bits.

3.2 Search Algorithm

Let us consider now the search algorithm for a pattern $P_{1 \dots m}$ [38–40]. We distinguish three types of occurrences of P in T, depending on the block layout (see Fig. 1):

1. the occurrence lies inside a single block (there are occ_1 occurrences of this type);
2. the occurrence spans two consecutive blocks, B_k and B_{k+1}, such that a prefix $P_{1 \dots i}$ matches a suffix of B_k and the suffix $P_{i+1 \dots m}$ matches a prefix of B_{k+1} (there are occ_2 occurrences of this type); and
3. the occurrence spans three or more blocks, $B_k \dots B_\ell$, such that $P_{i \dots j} = B_{k+1} \dots B_{\ell-1}$, $P_{1 \dots i-1}$ matches a suffix of B_k and $P_{j+1 \dots m}$ matches a prefix of B_ℓ (there are occ_3 occurrences of this type).

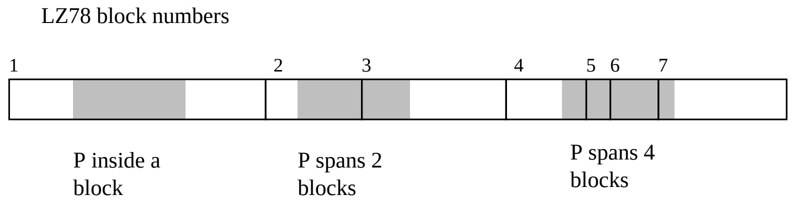


Fig. 1. Different situations in which P can match inside T.

Note that each of the $occ = occ_1 + occ_2 + occ_3$ possible occurrences of P lies exactly in one of the three cases above. We explain now how each type of occurrence is found.

Occurrences Lying Inside a Single Block. Given the properties of LZ78/ LZW, every block B_k containing P is formed by a shorter block B_ℓ concatenated to a letter c. If P does not occur at the end of B_k, then B_ℓ contains P as well. We want to find the shortest possible block B in the referencing chain for B_k that contains the occurrence of P. This block B finishes with the string P, hence it can be easily found by searching for P^r in *RevTrie*. Occurrences of type 1 are located in $O(m^2 \log \sigma + occ1)$ time.

Occurrences Spanning Two Blocks. P can be split at any position, so we have to try them all. The idea is that, for every possible split, we search for the reverse pattern prefix in *RevTrie* and the pattern suffix in *LZTrie*. Now we have two ranges, one in the space of reversed strings (i.e., blocks finishing with the first part of P) and one in that of the normal strings (i.e. blocks starting with the second part of P), and need to find the pairs of blocks $(k, k+1)$ such that k is in the first range and $k+1$ is in the second range. This is what the range searching data structure is for. Occurrences of type 2 are located in $O(m^3 \log \sigma + (m + occ2) \log n)$ time.

Occurrences Spanning Three Blocks or More. We need one more observation for this part. Recall that the LZ78/LZW algorithm guarantees that every block represents a different string. Hence, there is at most one block matching $P_{i...j}$ for each choice of i and j. This fact severely limits the number of occurrences of this class that may exist, $occ_3 = O(m^2)$. The idea is to identify maximal concatenations of blocks $P_{i...j} = B_k \dots B_\ell$ contained in the pattern, and thus determine whether B_{k-1} finishes with $P_{1...i-1}$ and $B_{\ell+1}$ starts with $P_{j+1...m}$. If this is the case we can report an occurrence. Occurrences of type 3 are located in $O(m^2 \log \sigma + m^3)$ time.

Overall, the query time is upper bounded by $O(m^3 \log \sigma + (m + occ) \log n)$.

3.3 Implementation of the Data Structures

The above data structures are built as follows [40, 39]. For the construction of *LZTrie* we traverse the text and at the same time build a *normal trie* (using one pointer per parent-child relation) of the strings represented by Ziv-Lempel blocks. At step k (assume $B_k = B_i \cdot c$), we read the text that follows and step down the trie until we cannot continue. At this point we create a new trie leaf (child of the trie node of block i, by character c, and assigning the leaf block number k), go to the root again, and go on with step $k+1$ reading the rest of the text. The process completes when the last block finishes with the text terminator "$". In Fig. 3 (upper left) we show the normal Ziv-Lempel trie for the text "`alabar_a_la_alabarda_para_apalabrarla$`".

Once we build the normal trie, we have enough information to build the final representation of *LZTrie*, using the parentheses representation of [36], with some practical considerations, such as representing the trie in its general tree form, instead of making it binary. To build the balanced parentheses representation we traverse the normal trie in preorder, writing an opening parenthesis each time a node is visited for the first time, traversing all subtrees of the node recursively in preorder, and then writing a closing parenthesis.

The *LZTrie* structure contains a sequence of parentheses representing the trie structure, a sequence *lets* of characters that label each edge of the trie, in preorder, and a sequence *ids* of block identifiers, also in preorder. In Fig. 3 (lower left) we show the balanced parentheses representation of *LZTrie* for the running example. We identify a trie node x with its opening parenthesis in the representation. The subtree of x contains those nodes (parentheses) enclosed between the opening parenthesis representing x and its matching closing parenthesis.

Once the *LZTrie* is built we free the space of the normal trie, and build *Node*. This is just an array with the n nodes of *LZTrie*, using $\lceil \log n \rceil$ bits for each. *Node* is constructed from the *ids* array in the following way. If the i-th position of the *ids* array belongs to the k-th block identifier, then the k-th position of Node stores the number i. In other words, *Node* is the *inverse* of permutation *ids*.

To construct *RevTrie* we traverse *LZTrie* in a depth-first order, generating each string stored in *LZTrie* in constant time, and then inserting it into a *normal trie of reversed strings*. For simplicity, the empty unary paths are not compressed in the normal trie. When we finish, we traverse the trie and represent *RevTrie* using a sequence of parentheses and block identifiers, *rids*. Empty unary nodes are removed only at this step, and so the number of nodes in *RevTrie* is $n \leqslant n' \leqslant 2n$. If we use $n' \log n$ bits for the *rids* array, in the worst case *RevTrie* requires $2uH_k(T) + o(u)$ bits of storage, and the whole index requires $5uH_k(T)(1+o(1))$ bits. Instead, we can represent the *rids* array with $n \log n$ bits (i.e., only the non-empty nodes), plus a bitmap of $2n(1+o(1))$ bits supporting *rank* queries in $O(1)$ time [1] [43]. The j-th bit of the bitmap is 1 if the node represented by the j-th open parenthesis is not an empty node, otherwise the bit is 0. The

[1] In this context $rank(j)$ is the number of 1's occurring before and including the j-th bit of the bitmap.

rids index corresponding to the j-th opening parenthesis is $rank(j)$. Using this representation, *RevTrie* requires $uH_k(T) + o(u)$ bits of storage. This was unclear in the original LZ-index paper [40, 39]. See Fig. 2 for an illustration.

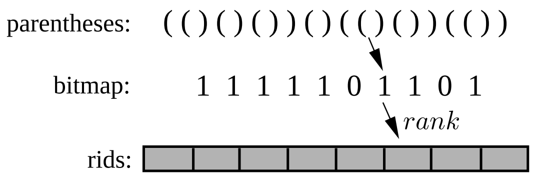

parentheses: (( ) ( ) ( )) ( ) (( ) ( )) (( ))
bitmap: 1 1 1 1 1 0 1 1 0 1
rank
rids:

Fig. 2. A $uH_k(T) + o(u)$ bits representation of *RevTrie*.

In practice, the *Range* data structure is replaced by *RNode*, a mapping from block identifiers to *RevTrie* nodes [40, 39]. *RNode* is built as the inverse of permutation *rids*.

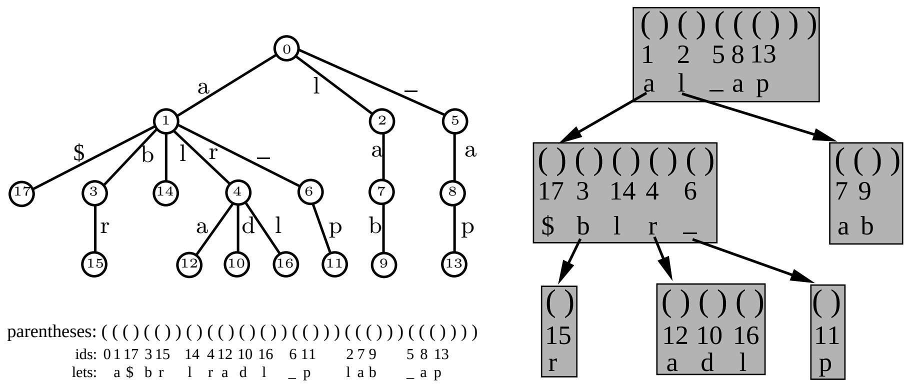
In the experiments of the original LZ-index [40, 39], the largest extra space needed to build *LZTrie* is that of the normal trie, which is 1.7–2.0 times the text size. The largest extra space to build *RevTrie* is that of the normal reverse trie, which is, in some cases, 4 times the text size. This is, mainly, because of the empty unary nodes. This space dictates the maximum indexing space of the algorithm. The overall indexing space was 4.8–5.8 times the text size for English text, and 3.4–3.7 times the text size for DNA. As a comparison, the construction of a plain suffix array without any extra data structure requires 5 times the text size. However, after we build the index we are left with a succinct representation, while a normal suffix array needs those 5 times the text size forever.

4 Thesis Proposal

In this section we present our thesis proposal. In the following subsections, for each part of the thesis we give an introduction, and then we define the objective, the main aspects of the proposed research, and expected results.

4.1 Space-efficient Construction of LZ-index

Many works on compressed full-text self-indexes do not consider the space-efficient construction of the indexes. For example, construction of *compressed suffix array* (*CS-array*) [44] and *FM-index* [9] involves building first the suffix array of the text. Similarly, the LZ-index is constructed over a non-compressed intermediate representation. In both cases, one needs about 5 times the text size. Thus, the final indexes require little working memory, but the memory required to build them may be excessive. For example, the Human Genome may fit in 1 Gb of main memory using these indexes (and thus it can be operated entirely in RAM on a desktop computer), but 15 Gb of main memory are needed to build



parentheses: ((( ) (( )) ( ) ((( ) ( ) ( )) (( ))) ((( ))) ((( ))))
ids: 0 1 17 3 15 14 4 12 10 16 6 11 2 7 9 5 8 13
lets: a $ b r l r a d l _ p l a b _ a p
( ) ( ) ((( )))
1 2 5 8 13
a l _ a p
( ) ( ) ( ) ( ) ( )
17 3 14 4 6
$ b l r _
(( ))
7 9
a b
( )
15
r
( ) ( ) ( )
12 10 16
a d l
( )
11
p

Fig. 3. Different representations of the Ziv-Lempel trie for the running example.

them! Using secondary memory for the construction is usually rather inefficient, thus we seek to avoid the use of secondary memory whenever possible.

The works of T.-W. Lam et al. [26] and W.-K.Hon et al. [16] deal with the space (and time) efficient construction of *CS-array*. The former work presents an algorithm that uses $(2H_0(T)+1+\epsilon)u$ bits of space to build the *CS-array*, where ϵ is any positive constant; the construction time is $O(\sigma u \log u)$, which is good enough if the alphabet is small (as in the case of DNA), but may be impractical in the case of proteins and Oriental languages, such as Chinese or Japanese. The second work [16] addresses this problem by requiring $(H_0(T) + 2 + \epsilon)u$ bits of main memory, and $O(u \log u)$ time to construct the *CS-array*. Also, they show how to build the *FM-index* from *CS-array* in $O(u)$ time.

The main memory requirement to build the LZ-index comes from the normal tries used to build *LZTrie* and *RevTrie*. In this part of the thesis, we aim at a practical and efficient algorithm to build those tries in little memory, by replacing the normal tries with space-efficient data structures that support insertions. These can be seen as hybrids between normal tries and the final parentheses representations.

We base the space-efficient construction of the tries in a representation of balanced parentheses [36], modified to allow a fast incremental construction as we traverse the text. In a linear sequence of balanced parentheses, the insertion of a new node at any position of the sequence may force rebuilding the sequence from scratch. To avoid that cost, we define a *hierarchical representation of balanced parentheses* (hrbp for short), such that we rebuild only a small part of the entire sequence to insert a new node. In a hrbp we cut the trie into *pages*, that is, into subsets of trie nodes such that if a node x is stored in page q, then node y, the parent of x, is: (1) also stored in q (enclosing x), or (2) stored in a page p, the *parent page* of q, and hence y is ancestor of all nodes stored in q. We store in p information indicating that node y encloses all nodes in q. In a hrbp we arrange the pages in a tree, thus the entire trie is represented by a tree of pages. A page is represented as a contiguous block of memory. In Fig. 3 (right) we show a hrbp for *LZTrie*. When we insert a new node in the hrbp (corresponding to a Ziv-Lempel block), we only need to recompute the page were the insertion is done.

Objective: To design, analyze, implement, and evaluate the empirical performance of an algorithm to construct LZ-index using little space.

Aspects to Consider in the Research:

– To define how much information is stored in each page of the hrbp, and how much is computed on the fly.
– To define a policy to achieve a minimum fill ratio α in the pages of the hrbp, thus controlling the wasted space.
– To define a method to solve *page overflows* (i.e., insertions in full pages). This method must minimize the number of pages in the hrbp, thus reducing the space wasted in pointers between pages.
– To compress empty unary paths when constructing *RevTrie*, and thus reducing even more the indexing space of this trie. To this end, we plan to use a *PATRICIA tree* [33] to represent *RevTrie*.
– To perform a theoretical analysis for construction time and indexing space of the algorithm.
– To perform an efficient implementation of a prototype of the space-efficient algorithm to construct the LZ-index.
– To use the prototype to obtain experimental results on the space-efficient construction of LZ-index. To compare against the method of W.-K.Hon et al. [16] and others.

Expected Results: A practical and space-efficient algorithm to construct LZ-index, such that the index can be used in many more practical situations. As in related works [26, 16], and using the properties of Ziv-Lempel compression, we hope to relate the space needed in the construction with the size of the compressed text. Also, we expect an indexing space close to that of the final index.

4.2 Reducing the Space Requirements of LZ-index

The space requirement of LZ-index is relatively large compared with competing schemes, such as *CS-array* and *FM-index*, which in practice require 0.6 to 0.7 and 0.3 to 0.8 times the text size respectively, versus 1.2 to 1.6 times the text size of LZ-index.

When we replace *Range* by *RNode* structure, the result is actually a "navigation" scheme that permits us moving back and forth from trie nodes to positions, both in *LZTrie* and *RevTrie*. The block identifiers are common to both tries and permit moving from one trie to the other.

Figure 4 shows the navigation scheme. Dashed arrows are "for free" in terms of memory, since they are followed by applying *rank*. The other four arrows are in fact the four main components in the space usage of the index: node identifiers in *LZTrie* (*ids*) and in *RevTrie* (*rids*), and tree nodes in *LZTrie* (*Node*) and in *RevTrie* (*RNode*). The structure is symmetric and we can move from any point to any other.

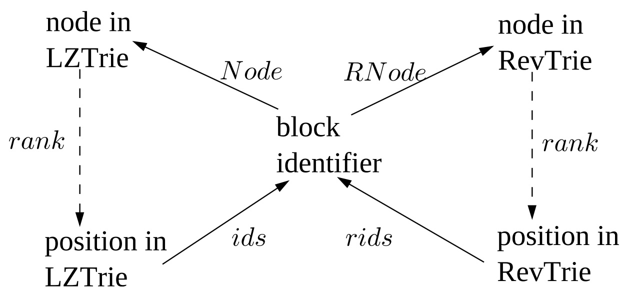


Fig. 4. The navigation structure over index components.

The structure, however, is redundant, in the sense that the number of arrows is not minimal. Given n nodes, $n-1$ arrows are sufficient to connect them in both directions (actually forming a ring structure). Figure 5 gives an alternative navigation scheme [37] where the minimum number of links are used. Note that arrays *rids* and *Node* have disappeared and have been replaced by mapping *Rev*, which given a rank position in the *RevTrie* directly gives the corresponding node in *LZTrie*, that is, $Rev(rpos) = Node(rth_r(rpos))$.

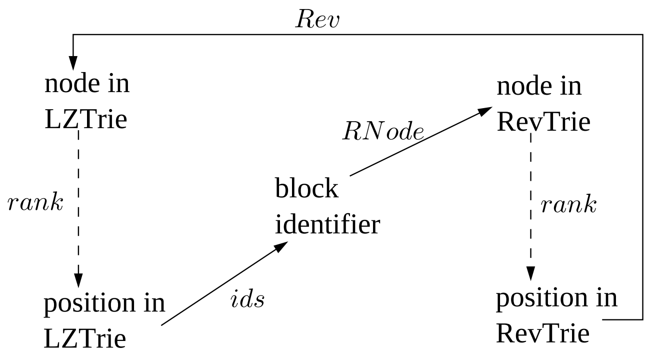


Fig. 5. The reduced navigation structure over index components.

The result is that the index works in about 3/4 of the space originally needed, at the expense of somewhat longer navigation paths in the query process. In some cases queries are even faster under this scheme, since the direct link *Rev* is faster than the direct application of $Node(rth_r(rpos))$. However, there are cases where we need to compute $Node(id)$, but as we do not have $Node$ now, we are forced to use $Rev(rank(RNode(id)))$ instead, which triples the cost. There are other cases where also the path length is 3. All these are needed for finding occurrences of type 2 and 3, and so the search time is increased.

In this part of the thesis we propose better ways to reduce the redundant information in the LZ-index with the aim of improving the space requirement of it. In Figure 6 we show a variant of the navigation scheme presented in Figure 5. In this new scheme we replace *RNode* by $rids^{-1}$, the inverse of permutation $rids$, and we add ids^{-1}, the inverse of permutation ids. However, the idea is not to store explicitly the inverse permutations. Instead, we use the idea of Munro et al. [35], which present a method requiring $(1+\epsilon)n\log n + O(1)$ bits of storage to represent a permutation and compute its inverse in $O(1/\epsilon)$ time, for a constant $0 < \epsilon < 1$. Note that we need $2n\log n$ bits to represent both permutations, and the time to compute a value of a permutation is $O(1)$. This new scheme has the advantage that the largest useful path length is 2. However, the cost of computing ids^{-1} and $rids^{-1}$ is not constant, and thus paths including these arrows have an additional cost. We hope that this new version will work also in about 3/4 of the space of the original LZ-index (plus the space of the data structures to compute the inverses), and that the search time will be competing with that of the original LZ-index.

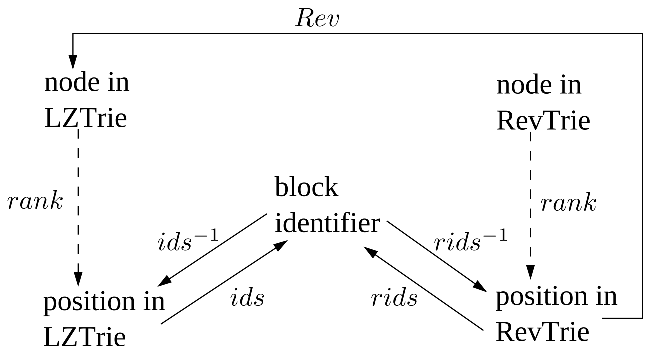


Fig. 6. A new variant of the navigation structure over index components.

Another navigation scheme is shown in Figure 7. In this case we replace the *Rev* array by a data structure to compute *select* (which is just the inverse of *rank*) in $O(1)$ time [34]. Given a position in *LZTrie*, *select* computes the corresponding *LZTrie* node. Thus, this version of the index requires less space (we hope about 0.6 times the size of the original LZ-index). It is important to note that the data structure of Munro et al. allows space/time trade-offs, and in this way we expect that this introduce space/time trade-offs to LZ-index.

Objective: To design, analyze, implement, and evaluate the empirical performance of different methods to reduce the space requirement of LZ-index, mainly

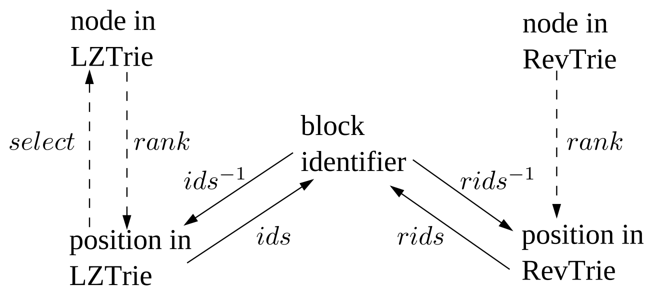


Fig. 7. Our navigation structure over index components.

by eliminating some of the redundancy in it, also providing space/time trade-offs to the index.

Aspects to Consider in the Research:

– To define the versions of LZ-index corresponding to the above schemes. We also plan to study the convenience of other navigation schemes.
– To define the construction and search algorithms on this new version of LZ-index. For the scheme of Figure 6, the search algorithm must to use the optimal paths in the navigation scheme, avoiding to use ids^{-1} or $rids^{-1}$ whenever exists an alternative path of the same length.
– To perform a theoretical analysis of working space and search time of this new alternatives. As the data structure to compute inverse permutations allows space/time trade-offs [35], we hope to obtain versions of LZ-index providing space/time trade-offs.
– To perform efficient implementations of prototypes of this new indexes.
– To use the prototypes to obtain experimental results of the space usage and search time of the new indexes, comparing against the original LZ-index and others.

Expected Results: To reduce the space requirements of LZ-index, and to get compressed full-text self-indexes providing a complete range of space/time trade-offs. We hope a version of LZ-index requiring about 0.6 times the space of the original index.

4.3 A Secondary Memory Prototype of LZ-index

Although compressed full-text self-indexes require little memory, there are cases where the text is so large that the corresponding self-index does not fit entirely in main memory. In these cases, the index must be stored in secondary storage, and the search proceeds by loading to main memory the relevant parts of the index. Because of its high cost, the problem here consists in reducing the number of accesses to secondary storage at search and construction time.

Remember that the initial statement in behalf of compressed full-text self-indexes was that larger texts could be indexed and stored in main memory, without accessing secondary memory. However, the advantage of using compressed full-text self-indexes on secondary storage is that the cost of transmission of the index from secondary to main memory can be reduced (recall *Advantage 2*, Section 1).

There do not exist many works on full-text indexes on secondary memory, which definitely is an important issue. One of the best known indexes for secondary memory is the *String B-tree* [8], although this is not a compressed data structure, requiring about 12 times the text size (not including the text) [7]. On the other hand, Clark and Munro [6] present a representation of suffix trees on secondary storage (the *Compact Pat Trees*, or *CPT* for short). They use a space-efficient representation of the trie structure, requiring $3u + o(u)$ bits of storage (i.e., this is not a compressed index, because the space requirement is not related to $H_k(T)$). Also, the index needs the text to operate. The representation is organized in such a way that the number of disk accesses is reduced to 3–4 per text search. The authors claim that the space requirement of their index is comparable to that of the suffix arrays, since it needs about 5–6 times the text size to operate. Finally, Mäkinen et al. [30] propose a technique to store the *CS-array* on secondary storage, based on *backward searching* [45]. This is the only proposal to store a compressed full-text self-index on secondary memory, requiring less than $u(H_0(T) + \log\log\sigma)$ bits of storage.

In this part of the thesis we propose to define a version of LZ-index that can be efficiently handled on secondary storage, both for constructing and searching. In this sense, the hrbp used in Subsection 4.1 to construct LZ-index can be useful, since it cuts the trie into pages which can be stored on secondary memory. Also it is a good idea to replace the *Node* and *RNode* mappings as in Subsection 4.2, modifying the technique of Munro et al. [35] to work on secondary storage. This reduces the secondary memory accesses when navigating from a trie to the other.

Objective: To design, analyze, implement, and evaluate empirical performance of an efficient version of LZ-index working on secondary memory.

Aspects to Consider in the Research:

- Definition of the data structures to effectively store the index on secondary memory. The trie operations (Subsection 3.1) must be efficiently implemented on this new representation.
- To define a construction algorithm that works on secondary memory, minimizing the navigation from *RevTrie* to *LZTrie* (when constructing *RevTrie*).
- To define a method that minimizes the navigation from *LZTrie* to *RevTrie* (and vice versa) at search time, reducing the accesses to secondary memory.
- To perform a theoretical analysis of space occupancy and number of secondary memory accesses at search and construction time.
- To perform an efficient implementation of a prototype of LZ-index on secondary memory.

– To use the prototype to obtain experimental results, comparing against other data structures working on secondary memory (such as *String B-trees* and *CPT*).

Expected Results: To obtain an efficient version of LZ-index working on secondary memory (both for constructing and searching).

4.4 A Dynamic LZ-index

Generally, in indexed text searching research, the text is modeled as a static sequence of characters. However, in real situations the insertion and deletion of parts of the text is rather common. This does not introduce major problems in sequential text searching scenarios, since there is not any data structure built on the text. However, in indexed text searching, the indexes must be updated upon text changes. This is currently a problem even on uncompressed full-text indexes, and not much has been done on this important issue.

Some works on dynamic full-text indexes are [8, 9, 17, 5], of which the last three are compressed self-indexes. Yet, those are very preliminary.

The model of the problem is the following. Let $\Delta = \{T_1, \ldots, T_s\}$ be a dynamic collection of texts having arbitrary lengths and total size u. Collection Δ may shrink or grow over time due to `insert` and `delete` operations which allow to add or remove from Δ an individual text string.

It is important to note that the intermediate hrbp of the tries in Subsection 4.1 can be made searchable, so that it could be taken as the final index. The result would be a LZ-index supporting efficient insertion of new text, since it can be seen as the insertion of a new text T_{s+1} at the end of Δ. The problem here arises with the deletion of text, since it can be performed at any part of the collection.

Objective: To design, analyze, implement, and evaluate empirical performance of a version of LZ-index allowing insertions and deletions of text (both on main and secondary memory).

Aspects to Consider in the Research:

– To define an algorithm to allow efficient insertion of text to LZ-index. We plan to use the hrbp of Subsection 4.1 to avoid recomputing the whole index on insertions of text.
– To perform the theoretical analysis for the cost of updating the index upon insertions.
– To perform an efficient implementation of a prototype of this semi-dynamic LZ-index.
– To use the prototype to obtain experimental results on the viability of the method in practice.
– To define, on the above representation, a deletion algorithm such that the index can be updated on deletions of part of the text, in order to get a fully-dynamic index.

– To perform a theoretical analysis of the cost of updating the index on deletions.
– To perform an efficient implementation of the deletion algorithm on the semi-dynamic prototype, thus obtaining a fully-dynamic one.
– To use the prototype to obtain experimental results on dynamic texts both in main and secondary memory.

Expected Results: An efficient version of LZ-index, which can be efficiently updated on text changes. We hope that the results will be of independent interest for dinamizing other Ziv-Lempel schemes.

5 Deliverables

– The main contributions of our thesis will be new theoretical developments in the area of compressed full-text self-indexes based on the Ziv-Lempel compression algorithm.
– We hope to publish our main results in at least four high-level international conferences, and in two ISI Journals.
– We plan to implement prototypes of our algorithms, which will be publicly available such that they can be used by the scientific community as well as practitioners looking for particular solutions to practical problems in Computational Biology, Digital Libraries, and full-text databases in general.

6 Work One on Advance

We have already worked on the goals of Section 4.1, that is, the space-efficient construction of LZ-index. As a result we have obtained a space-efficient and practical construction algorithm with the following main properties:

– Allowing to choose a minimum fill ratio α in the pages of the hrbp, $0 < \alpha < 1$, and with a method to solve page overflows that minimizes the number of pages in the hrbp.
– Requiring $(4 + \epsilon)uH_k(T) + o(u)$ bits to construct LZ-index in $O(\sigma u)$ time (recall that the final index requires $4uH_k(T)(1 + o(1))$ bits of storage).
– In practice, by choosing appropriately the value of α, the indexing space is close to that of the final index, as predicted by the theoretical analysis. That is, whenever the LZ-index can be used, we can build it.
– The indexing speed is approximately 5 sec/Mb (in a 2GHz machine), which seems much better than competing schemes [26, 16] (although we have not yet directly compared our method to those approaches).

All these results have been submitted to the *16th Annual International Symposium on Algorithms and Computation* (*ISAAC 2005*), whose proceedings are published by *Springer* in the *Lecture Notes in Computer Science* series. We attach to this proposal the submitted paper.

7 Conclusions

The current trend in indexed full-text searching is that of compressed full-text self-indexes, which replace the text with a more space-efficient representation of it, and at the same time this representation provides indexed access to the text. As a consequence, larger texts can be indexed and stored in main memory.

There is much work on indexed text searching, and many goals have been obtained separately. In this proposal we have defined the working plan for our thesis. As a general objective we hope to contribute in the track of compressed full-text self-indexes based on Ziv-Lempel compression. Our specific objective is to add many interesting features to the LZ-index compressed full-text self-index [38–40], obtaining an index with the following properties: fast full-text searching and text recovery; using little space for construction and operation; allowing insertion and deletion of text; providing a range of space/time trade-offs; and efficient construction and search in secondary memory.

References

1. M. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. SPIRE'02*, LNCS 2476, pages 31–43, 2002.
2. A Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
3. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice–Hall, 1990.
4. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
5. H.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. CPM'04*, LNCS 3109, pages 445–456, 2004.
6. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. SODA'96*, pages 383–391, 1996.
7. P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *Proc. SODA'96*, pages 373–382. SIAM, 1996.
8. P. Ferragina and R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.
9. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc FOCS'00*, pages 390–398, 2000.
10. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pages 269–278, 2001.
11. P. Ferragina and G. Manzini. On compressing and indexing data. Technical Report TR-02-01, Dipartamento di Informatica, Univ. of Pisa, 2002.
12. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc.SPIRE'04*, LNCS 3246, pages 150–160. Springer, 2004.
13. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850. SIAM, 2003.
14. R. Grossi, A. Gupta, and J.S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, pages 636–645. SIAM, 2004.

15. R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pages 397–406, 2000.
16. W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. ISAAC'03*, LNCS 2906, pages 240–249, 2003.
17. W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yu. Compressed index for dynamic text. In *Proc. DCC'04*, pages 102–111, 2004.
18. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS'03*, pages 251–260, 2003.
19. J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. CPM'95*, LNCS 937, pages 191–204, 1995.
20. J. Kärkkäinen. *Repetition-based text indexes.* PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999.
21. J. Kärkkäinen and S. Rao. Full-text indexes in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume LNCS 2625, chapter 7, pages 149–170. Springer-Verlag Berlin, 2003.
22. J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. WSP'96*, pages 141–155. Carleton University Press, 1996.
23. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, LNCS 1090, pages 219–230, 1996.
24. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
25. S. Kurtz. Reducing the space requeriments of suffix trees. Technical Report 98-03, Technische Kakultät, Universität Bielefeld, Germany, 1998.
26. T.-W. Lam, K. Sadakane, W.-K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. COCOON 2002*, pages 401–410, 2002.
27. V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.
28. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, LNCS 3109, pages 420–433, 2004.
29. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, LNCS 3537, pages 45–56, 2005.
30. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. ISAAC'04*, LNCS 3341, pages 681–692. Springer, 2004.
31. U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
32. G. Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
33. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
34. I. Munro. Tables. In *Proc. FSTTCS'96*, LNCS 1180, pages 37–42. Springer, 1996.
35. I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *ICALP*, LNCS 2719, pages 345–356, 2003.
36. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS'97*, pages 118–126, 1997.
37. G. Navarro. Implementing the LZ-index: Theory versus practice. Unpublished personal note.

38. G. Navarro. Indexing text using the Ziv-Lempel trie. In *Proc. SPIRE'04*, LNCS 2476, pages 325–336, 2002.
39. G. Navarro. Indexing text using the Ziv-Lempel trie. Technical Report TR/DCC-2002-2, Dept. of Computer Science, Univ. of Chile, 2002. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/lzindex.ps.gz`.
40. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
41. G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
42. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.
43. V. Raman and S. Rao. Static dictionaries supporting rank. In *Proc. ISAAC '99*, LNCS 1741, pages 18–26, 1999.
44. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pages 410–421, 2000.
45. K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. SODA'02*, pages 225–232, 2002.
46. T. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, 1984.
47. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, New York, second edition, 1999.
48. J. Ziv and A. Lempel. Compression of individual sequences via variable–rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

Diego Arroyuelo
PhD Student

Gonzalo Navarro
Advisor