Space-Efficient Construction of LZ-Index*

Diego Arroyuelo and Gonzalo Navarro

Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, Santiago, Chile
{darroyue, gnavarro}@dcc.uchile.cl

Abstract. A *compressed full-text self-index* is a data structure that replaces a text and in addition gives indexed access to it, while taking space proportional to the compressed text size. The LZ-index, in particular, requires $4uH_k(1 + o(1))$ bits of space, where u is the text length in characters and H_k is its k-th order empirical entropy. Although in practice the LZ-index needs 1.0-1.5 times the text size, its construction requires much more main memory (around 5 times the text size), which limits its applicability to large texts. In this paper we present a practical space-efficient algorithm to construct LZ-index, requiring $(4+\epsilon)uH_k + o(u)$ bits of space, for any constant $0 < \epsilon < 1$, and $O(\sigma u)$ time, being σ the alphabet size. Our experimental results show that our method is efficient in practice, needing an amount of memory close to that of the final index.

1 Introduction and Previous Work

A *full-text database* is a system providing fast access to a large mass of textual data. The simplest (yet realistic and rather common) scenario is as follows. The text collection is regarded as a unique sequence of characters $T_{1...u}$ over an alphabet Σ of size σ, and the search pattern $P_{1...m}$ as another (short) sequence over Σ. Then the text search problem consists of finding all the occ occurrences of P in T. To provide fast access, data structures called *indexes* are built on the text. Typical text databases contain natural language texts, DNA or protein sequences, MIDI pitch sequences, program code, etc.

Until a short time ago, the smallest indexes available in practice were the suffix arrays [21], requiring $u \log u$ bits ($\log$ means $\log_2$ in this paper). 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). To handle huge texts like the Human Genome (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.

Several attempts to reduce the space of the suffix trees [2] or arrays [13,17,19,1] focused on reducing the size of the data structures but not the text, and did not relate text compressibility with the size of its index.

A parallel track started at about the same time [15,14,10,28,4,5,6,8,9,20,7], with the distinguishing feature of providing *compressed* indexes, whose sizes are proportional to the compressed text size. Moreover, in most cases, those indexes *replace* the text by

* Supported in part by CONICYT PhD Fellowship Program (first author) and Fondecyt Grant 1-050493 (second author).

being able to reproduce any text substring. This is called *self-indexing*. Taking space proportional to the compressed text, replacing it, and providing efficient indexed access to it, is an unprecedented breakthrough in text indexing and compression.

The LZ-index [24,25,26] is a full-text self-index on these lines, based on the Ziv-Lempel parsing of the text. If the text is parsed into n *phrases* by the LZ78 algorithm [29], 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. $4uH_k + o((1 + H_k)u)$, for any $k = o(\log n/ \log^2 \sigma)$ [16,6]. See the original article for details on its search algorithms, as we focus only in construction in this paper.

However, all these works do not consider the space-efficient construction of the indexes. For example, construction of *CS-array* [28] and *FM-index* [4] 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. 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 them! Using secondary memory for the construction is usually rather inefficient.

The works of T.-W. Lam et al. [18] and W.-K.Hon et al. [12] deal with the space (and time) efficient construction of *CS-array*. The former work presents an algorithm that uses $(2H_0 + 1 + \epsilon)u$ bits of space to build the *CS-array*, where H_0 is the 0-th order empirical entropy of the text, and 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 [12] addresses this problem by requiring $(H_0 + 2 + \epsilon)u$ bits of space and $O(u \log u)$ time to build the *CS-array*. Also, they show how to build the *FM-index* from *CS-array* in $O(u)$ time.

Our work follows this line of research. We present a practical and efficient algorithm to construct the LZ-index using little space. Our idea is to replace the (non-compressed) intermediate representations of the tries that conform the index by space-efficient counterparts. Basically, we use the balanced parentheses representation of Munro and Raman [23] as an intermediate representation for the tries, but we modify such representation to allow fast incremental construction directly from the text. The resulting intermediate data structure consists of a tree whose nodes are small subsequences of balanced parentheses, which are easier and cheaper to update. The idea is inspired in the work of Clark and Munro [3], yet ours differs in numerous technical aspects and practical considerations (structuring inside the nodes, overflow management policies, etc.).

Our algorithm requires $(4+\epsilon)uH_k+o(u)$ bits to build the LZ-index, for any constant $0 < \epsilon < 1$. This is very close to the space the final LZ-index requires to operate. This is the *first* construction algorithm for a self-index requiring space proportional to H_k instead of H_0. In practice our algorithm also requires about the same memory as the final index. That is, wherever the LZ-index can be used, we can build it. The indexing speed is approximately 5 sec/MB in a 2GHz machine, which is competitive with the (non-space-efficient) construction of *FM-index* and much faster than *CS-array* construction [26]. We argue that our method outperforms (in time) previous work [12] when indexing the Human Genome, using about the same indexing space.

2 The LZ-Index Data Structure

Assume that the text $T_{1...u}$ has been partitioned using the LZ78 [29] algorithm into $n+1$ blocks $T = B_0 \dots B_n$, such that $B_0 = \varepsilon$; $\forall k \neq \ell$, $B_k \neq B_\ell$; and $\forall k \geqslant 1$, $\exists \ell < k$, $c \in \Sigma$, $B_k = B_\ell \cdot c$. To ensure that B_n is not a prefix of another B_i, we append to T a special character "\$" $\notin \Sigma$. The data structures that conform LZ-index are [26,25]:

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.
2. *RevTrie*: is the trie formed by all the reverse strings $B_0^r \dots B_n^r$. In this trie there could be internal nodes not representing any block. We call these nodes *empty*.
3. *Node*: is a mapping from block identifiers to their node in *LZTrie*.
4. *RNode*: is a mapping from block identifiers to their node in *RevTrie*.

Each of these 4 structures requires $n \log n(1 + o(1)) = uH_k(1 + o(1))$ bits of space.

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 "\$".

Then we compact the normal trie, essentially using the parentheses representation of Munro and Raman [23]. We traverse the normal trie in preorder, writing an opening parenthesis when going down to a node, and a closing parenthesis when going up.

The *LZTrie* structure consists of the above sequence of parentheses plus a sequence *lets* of characters that label each trie edge and a sequence *ids* of block identifiers, both in preorder. 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. It is built as the inverse of permutation *ids*.

To construct *RevTrie* we traverse *LZTrie* in depth-first order, generating each string stored in *LZTrie* in constant time, and then inserting it into a *normal trie of reversed strings*. We then traverse the trie and represent it using a sequence of parentheses and block identifiers, *rids*. Empty unary nodes are removed only at this step. Finally, we build the normal reverse trie and build *RNode* as the inverse permutation of *rids*.

In the experiments of the original LZ-index [26,25], 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 indexing space for the normal reverse trie 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 (note that the text itself can be processed by buffers and hence does not require significant space). 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.

3 Space-Efficient Construction of LZTrie

The main memory requirement to build the LZ-index comes from the normal tries used to build *LZTrie* and *RevTrie*. We focus on building those tries in little memory, by replacing them with space-efficient data structures that support insertions. These can be seen as hybrids between normal tries and their final parentheses representations.

Let us start with *LZTrie*. In its final representation as a linear sequence of balanced parentheses [23], the insertion of a new node at any position of the sequence may force rebuilding the sequence from scratch. To avoid that cost, we work on 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 in *pages*, that is, in 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.

We represent a page as a contiguous block of memory. Let $N_1 < \ldots < N_t$ be even integers. We say that a page has size N_i if it can store N_i parentheses ($N_i/2$ nodes), although its physical size is larger than N_i bits. Each page p of size N_i consists of: N_i bits to represent a subsequence of balanced parentheses; $N_i/2$ bits (the *flags*) indicating which opening parentheses (nodes) in a page have their subtree stored in a child page; $\lceil \log N_i/2 \rceil$ bits to tell the number of nodes stored in the page; $(N_i/2)\lceil \log u \rceil$ bits to store the block identifiers (*ids*) in the page (in preorder); $(N_i/2)\lceil \log \sigma \rceil$ bits to store the characters (*lets*) in the page (in preorder); and a variable number of pointers to child pages. The number of pointers varies from 0 to $N_i/2$, and it corresponds to the number of flags with value 1 in p. To maintain a constant physical page size, these pointers are stored in a separately allocated array, and we store a pointer to them in the page.

As in the parentheses representation of *LZTrie*, in the hrbp a node encloses its subtree, but not necessarily a node and its subtree are stored both in the same page. If the subtree of the j-th opening parenthesis of page p is stored in page q, then q is a child page of p and the j-th flag in p has the value 1. If the number of flags in 1 before the j-th flag (not including it) is h, then the h-th pointer of p points to q. An important property we enforce is that sibling nodes must be stored all in the same page.

Initially, the data structure consists of a sole empty page (the *root page*) of size N_1. The construction of *LZTrie* proceeds as explained in Section 2, but this time the nodes are inserted in a hrbp of *LZTrie*, instead of a normal trie. The insertion of a new node $B_k = B_i \cdot c$ in the hrbp implies to recompute the page in which the insertion is done. If the new leaf must become j-th opening parenthesis in the page (counting from left to right), then we insert "`()`" at the corresponding parentheses position and the j-th flag is set to 0. Also, c is inserted at position j within the characters, and k is inserted at the same position within the identifiers.

We do not store information to traverse the parentheses sequence in the pages of the hrbp. Instead, all the navigation inside each page is done sequentially, in a single $O(N_t)$ time pass: the first child of an opening parenthesis starts at the next position (unless that contains a closing parenthesis, in which case the node is a leaf in the page), and the next sibling starts right after the closing parenthesis matching the current position, which is

found sequentially. As we traverse the page, we maintain the current position j in *flags*, *ids* and *lets*, as well as the count h of 1-bits seen in *flags*.

A *page overflow* occurs when inserting a new node in a full page p. If the size of p is N_i, $1 \leqslant i < t$, we perform a `grow` operation on p, which allocates a new page p' of size N_{i+1}, copies the content of p to p', frees p, and replaces the pointer to p by a pointer to p' in the parent of p. If the size of p is N_t, instead, we select a subset of nodes to be copied to a new child page of p and then deleted from p.

We only allow the selection of the whole subtree of a node in the page (without selecting the node itself). This simple way ensures that sibling nodes are always stored in the same page. As the maximum number of siblings is σ, we must have $N_t \geqslant 2\sigma$ so that a page with children always has space for its top-level nodes at least. We choose the subtree of maximum size not exceeding $N_t/2$ nodes. It is easy to see that this guarantees that the size of the new leaf p' is at least $\lfloor N_t/(2\sigma) \rfloor - 1$ nodes.

Assume we have selected in this way the subtree of the j-th opening parenthesis in the page. The selected subtree is copied to a new page p', whose size is the minimum N_i suitable to hold the subtree. As p' is a new leaf page, all its flags are initialized to 0. Next we add in p a pointer to p', inserted at the current (j-th) position, and set to 1 the j-th bit in *flags*. Finally, we delete from p the selected subtree. After that, if the number of parentheses in p does not exceed N_i for some $i < t$, we perform a `shrink` operation, which is the opposite of `grow`.

Once we solved the overflow, the insertion of the new node may have to be done in p or in p', but we are sure that there is room for the new pair of parentheses in either page. The following lemma states the condition to achieve a minimum fill ratio α in the pages of the data structure, thus controlling the wasted space. The proof is obvious.

Lemma 1. *Let $0 < \alpha < 1$ be a real number. If each page has the smallest posible size N_i to hold its parentheses, and we define $N_i = N_{i-1}/\alpha$, $i = 2, \ldots, t$, and $2 \leqslant N_1 \leqslant 2/\alpha$, then all pages of the data structure have a fill ratio of at least α.*

As the trie has n nodes, we need $2n + n + n \log u + n \log \sigma + n \log u(2\sigma/N_t)$ bits of storage to represent the parentheses, flags, identifiers, characters and pointers to child pages, respectively. The last bound holds because leaves are created with at least $N_t/(2\sigma)$ nodes and thus there is at worst one pointer for each $N_t/(2\sigma)$ nodes (except the root). If, in addition, we define the N_is as in Lemma 1, in the worst case the construction algorithm needs $\frac{n}{\alpha}(3 + \log \sigma + (1 + 2\sigma/N_t) \log u)$ bits of storage. We can relate this space requeriment to H_k: as $n \log u = uH_k + O(kn \log \sigma)$ for any k [16], and since $n \leqslant u/\log_\sigma u$, the space is $\frac{1+2\sigma/N_t}{\alpha} uH_k + o(u)$ for any $k = o(\log n/\log^3 \sigma)$.

When constructing *LZTrie*, the navigational cost per character of the text is $O(N_t)$ in the worst case. Hence, the overall navigational cost is $O(N_t u)$. On the other hand, the cost of rebuilding pages after an insertion is $O(N_t)$, with or without page overflows. As there are n insertions, the total cost is $O(N_t n)$. However, the constant involved with page overflows is greater than that of simple insertions, thus in practice we expect that larger values of α yield a greater construction time (and a smaller space requirement). In general, choosing $N_t = 2\sigma/\gamma$ for any constant $0 < \gamma < 1$, we get $\frac{1+\gamma}{\alpha} uH_k + o(u)$ bits of space, which can be made $(1 + \epsilon)uH_k + o(u)$ for any constant $0 < \epsilon < 1$ by properly choosing γ and α. The construction time is $O(\frac{1}{\gamma}\sigma u) = O(\sigma u)$.

Once we construct the hrbp for *LZTrie*, we build the final version of *LZTrie* in $O(n)$ time. We perform a preorder traversal on the hrbp, writing an opening parenthesis each time we reach a node, then checking the corresponding flag, traversing the subtree of the node recursively in preorder (which, depending of the flag, may be stored in the same or in a child page), and then writing a closing parenthesis.

4 Space-Efficient Construction of *RevTrie*

For the space-efficient construction of *RevTrie*, we use a hrbp to represent not the original reverse trie but its *PATRICIA tree* [22], which compresses *empty* unary paths of the reverse trie. This yields an important saving of space. We do not store the skips in each node since they can be computed using the connection with *LZTrie*. We store, in the nodes of the reverse trie, pointers to nodes of *LZTrie*, instead of the corresponding block identifiers. Each pointer uses $\lceil \log 2n \rceil$ bits. This is done to avoid access to *Node* when constructing the reverse trie, so we can build *Node* after both tries have been built (thus reducing the indexing space). The empty non-unary nodes are marked by storing in them the same pointer to *LZTrie* stored in their first child.

To construct the reverse trie we traverse *LZTrie* in depth-first order, generating each string B_i stored in *LZTrie* in constant time, and then inserting its reverse B_i^r into the reverse trie. As we compress empty unary paths, the edges of the trie are labeled with strings instead of simple characters. The *PATRICIA* tree stores only the first character of the string that labels the edge. Given a node v in the reverse trie, the position of the character in B_i^r on which v discriminates is 1 plus the length of the string represented by v. If v is not an empty node, then it stores a pointer to *LZTrie* node n_v. The length of the string is the same as the depth of n_v in *LZTrie*, which can be computed in constant time [26]. On the other hand, if v is an empty node, we we must use instead a procedure similar to that used in [26] to compute the string that labels an edge of the trie.

The hrbp of the reverse trie requires at least $\frac{1}{\alpha}(2n' + n' + n' \log 2n + n' \log \sigma + (2\sigma/N_t)n' \log n')$ bits of storage to represent the parentheses, flags, pointers to *LZTrie*, characters and pointers to child pages, respectively, where $n' \geqslant n$ is the number of nodes in the reverse trie. As we compress unary paths, $n' \leqslant 2n$, and thus the space is upper bounded by $\frac{2(1+2\sigma/N_t)}{\alpha} uH_k + o(u)$. However, in practice we expect that n' will be much less than $2n$ (see Section 5 for empirical results).

For each string B_i^r to be inserted in the reverse trie, $1 \leqslant i \leqslant n$, the navigational cost is $O(N_t|B_i^r| + |B_i^r|^2)$ in the worst case (when we work $O(N_t)$ per character, and every traversed node is empty). The total construction cost is $\sum_{i=1}^{n} (N_t|B_i^r| + |B_i^r|^2)$. The sum $\sum_{i=1}^{n} |B_i^r|^2$ is usually $O(u \log_\sigma u)$, but in pathological cases it is $O(u^{3/2})$. To have a better theoretical bound, we can explicitly store the skips, using $2 \log \log u$ extra bits per node (inserting *empty* unary nodes when the skip is exceeded). In this way, one of each $\log^2 u$ empty unary nodes could be explicitly represented. In the worst case there are $O(u)$ empty unary nodes, of which $O(\frac{u}{\log u})$ are explicitly represented. This means $o(u)$ extra bits in the hrbp, and the construction cost is reduced to $\sum_{i=1}^{n} (N_t|B_i^r| + |B_i^r|)$. As $\sum_{i=1}^{n} |B_i^r| = u$, the total cost is $O(N_t u)$.

After we construct the hrbp for the reverse trie, we construct *RevTrie* directly from the hrbp in $O(n')$ time, replacing the pointers to *LZTrie* by the corresponding block

identifiers (*rids*), and then we free the space of the hrbp. If we use $n' \log n$ bits for the *rids* array, *RevTrie* requires $2uH_k + o(u)$ bits of storage, and the whole index requires $5uH_k(1+o(1))$ 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 [27]. 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 *rids* index corresponding to the j-th opening parenthesis is $rank(j)$. Using this representation, *RevTrie* requires $uH_k + o(u)$ bits of storage. This was unclear in the original LZ-index paper [26,25].

We finally build *Node* mapping from *ids* array in time $O(n)$ and $n \log n = uH_k + o(u)$ bits of space, and *RNode* from *rids* in $O(n')$ time and $n \log n' = uH_k + o(u)$ bits.

Now we summarize the construction process, and show in parentheses the total space requeriment and the time in each step. Then, we conclude with a theorem.

1. We build the hrbp of *LZTrie* from the text ($(1 + \epsilon)uH_k + o(u)$ bits, $O(\sigma u)$ time).
2. We build *LZTrie* from its hrbp ($(1 + \epsilon)uH_k + uH_k + o(u)$ bits, $O(n)$ time).
3. We free the hrbp of *LZTrie* and build the hrbp of the reverse trie from *LZTrie* ($(2 + \epsilon)uH_k + uH_k + o(u)$ bits, $O(\sigma u)$ time).
4. We build *RevTrie* from its hrbp ($(2+\epsilon)uH_k + uH_k + uH_k + o(u)$ bits, $O(n)$ time).
5. We free the hrbp of *RevTrie* and build *Node* from *ids* ($uH_k + uH_k + uH_k + o(u)$ bits, $O(n)$ time).
6. We build *RNode* from *rids* ($uH_k + uH_k + uH_k + uH_k + o(u)$ bits, $O(n)$ time).

Theorem 1. *Our space-efficient algorithm to construct LZ-index requires* $(4+\epsilon)uH_k + o(u)$ *bits of space, reached at step 4 above, and* $O(\sigma u)$ *time. This holds for any constant* $0 < \epsilon < 1$ *and any* $k = o(\log n / \log^3 \sigma)$.

5 Experimental Results

For the experiments we use the file `ohsumed.88-91` from the *OHSUMED* collection [11], as a representative of other text collections we tested, such as DNA, music, and others. We use incremental subsets of the file, ranging from 10MB to 100MB. We run our experiments on an AMD Athlon processor at 2GHz, 1024MB of RAM and 512Kb of cache, running version 2.6.7-gentoo-r11 of Linux kernel. We compiled the code with `gcc 3.3.4` using optimization option `-O9`. Times were obtained using 10 repetitions.

In Fig. 1 we show the construction space for *LZTrie* and *RevTrie*. As expected, the construction space is smaller as we use a greater value of α. On average, the construction space of *LZTrie* ranges from approximately 0.5 ($\alpha = 0.95$) to 0.64 ($\alpha = 0.5$) times the text size, and from approximately 1.00 ($\alpha = 0.95$) to 1.27 ($\alpha = 0.5$) times the size of the final version of *LZTrie*. For construction of *RevTrie* the space varies from 0.52 ($\alpha = 0.95$) to 0.65 ($\alpha = 0.5$) times the text size, and from 1.47 ($\alpha = 0.95$) to 1.85 ($\alpha = 0.5$) times the size of the final *RevTrie*. The greater difference among *RevTrie* and its hrbp is due to the fact that the final version of the trie does not store the characters. As a comparison, the original construction algorithm [26] (labeled “Original” in the plots) needs on average 1.82 times the text size to hold the normal trie and 3.30 times to hold the normal reverse trie. The sizes as a fraction of the final tries are 3.62 and 9.87 times, respectively.

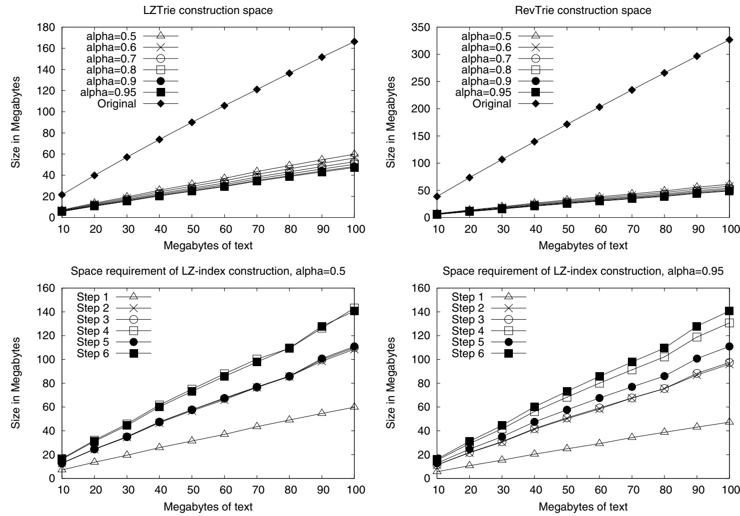

Fig. 1. Size of the hrbps of *LZTrie* and *RevTrie*, $N_1 = 2$, N_t = 512

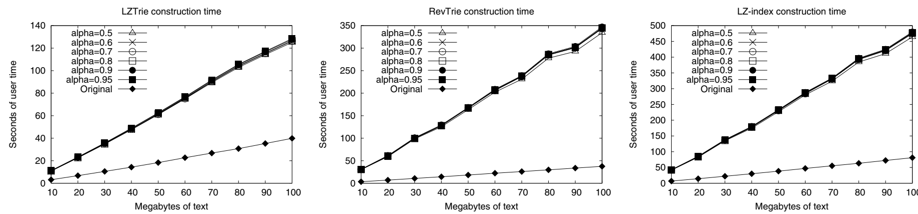

Fig. 2. Average user time to build *LZTrie*, *RevTrie* and the whole LZ-index, $N_1 = 2$, N_t = 512

In the same Fig. 1 (below) we show the space requirements in each step of the construction. The space to construct LZ-index varies from 1.46 ($\alpha = 0.95$) to 1.49 ($\alpha = 0.5$) times the text size, and from 1.00 ($\alpha = 0.95$) to 1.02 ($\alpha = 0.5$) times the size of the final index (labeled "Step 6" in the plots). This confirms that the indexing space is about the same to that needed by the final index. For $\alpha = 0.5$ the maximum is reached in step 4, as predicted in the analysis. However, for $\alpha = 0.95$ the maximum is reached in step 6, mainly because in the experiments the number of nodes of the reverse trie is (on average) $n' \approx 1.032n$, which is much less than the pesimistic theoretic bound $n' \leqslant 2n$ we used in the space requirement analysis.

In Fig. 2 we show the indexing time for the tries and the whole index. The average indexing rate for *LZTrie* varies from 0.805MB/sec ($\alpha = 0.95$) to 0.828MB/sec ($\alpha = 0.5$). For *RevTrie* it varies from 0.302MB/sec ($\alpha = 0.95$) to 0.309MB/sec ($\alpha = 0.5$). The whole indexing rate varies from 0.217MB/sec ($\alpha = 0.95$) to 0.223MB/sec ($\alpha = 0.5$). As we expected, the indexing rate is greater as α becomes smaller. The original construction has an average indexing rate of 2.745MB/sec for *LZTrie*, 2.752MB/sec for *RevTrie*, and 1.310MB/sec for the whole indexing process. Thus the succinct construction is 6 times slower in practice, as the upper bound $O(\sigma u)$ is too pessimistic.

We also tested our algorithm on DNA data [1], where the indexing rate is about 0.197MB/sec ($\alpha = 0.95$, $N_1 = 2$, $N_t = 192$), using on average 1.19 times the text size of main memory to index. Extrapolating these results we can argue that the human genome can be indexed in approximately 4.23 hours and using less than 4 GB of main memory. As a comparison, W.-K. Hon et al. [12] argued that they can index the human genome in 20 hours (although they do not describe the CPU of the machine used).

6 Conclusions and Future Work

In this paper we proposed a practical space-efficient algorithm to construct LZ-index. The basic idea is to construct the tries of LZ-index using space-efficient intermediate representations that allow fast incremental insertion of nodes. The algorithm requires at most $(4+\epsilon)uH_k + o(u)$ bits ($0 < \epsilon < 1$) to construct LZ-index for the text $T_{1\ldots u}$ in time $O(\sigma u)$, being σ the alphabet size. This is the first construction algorithm of a compressed full-text index whose space requirement is related to H_k (the k-th order empirical entropy of the text). In our experiments the construction required approximately 1.45 times the text size, or 1.02 times the final index size, which is much better than the original LZ-index construction algorithm (4–5 times the text size), and the indexing speed was approximately of 5sec/MB.

We believe that our intermediate hrbp representation could 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. Those pages could also be handled in secondary memory, so as to have an efficient disk-based LZ-index. Furthermore, the hrbp might have independent interest as a practical technique to represent dynamic general trees in little space, so we plan to work on making them fully dynamic. For the near future, we plan to compare our method against previous work [12], both in time and space.

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. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. SODA'96*, pages 383–391, 1996.
4. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc FOCS'00*, pages 390–398, 2000.
5. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pages 269–278, 2001.
6. P. Ferragina and G. Manzini. On compressing and indexing data. Technical Report TR-02-01, Dipartamento di Informatica, Univ. of Pisa, 2002.
7. 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.
8. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850. SIAM, 2003.

[1] 52.71MB from *GenBank* (Homo Sapiens DNA, `http://www.ncbi.nlm.nih.gov`).

9. 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.
10. 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.
11. W. Hersh, C. Buckley, T. Leone, and D. Hickam. Ohsumed: An interactive retrieval evaluation and new large test collection for research. In *Proc. SIGIR'94*, pages 192–201, 1994.
12. 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.
13. J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. CPM'95*, LNCS 937, pages 191–204, 1995.
14. J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999.
15. 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.
16. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
17. S. Kurtz. Reducing the space requeriments of suffix trees. Technical Report 98-03, Technische Kakultät, Universität Bielefeld, Germany, 1998.
18. 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.
19. V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.
20. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, LNCS 3537, pages 45–56, 2005.
21. U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
22. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
23. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. FOCS'97*, pages 118–126, 1997.
24. G. Navarro. Indexing text using the Ziv-Lempel trie. In *Proc. SPIRE'04*, LNCS 2476, pages 325–336, 2002.
25. 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`.
26. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
27. V. Raman and S. Rao. Static dictionaries supporting rank. In *Proc. ISAAC '99*, LNCS 1741, pages 18–26, 1999.
28. 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.
29. J. Ziv and A. Lempel. Compression of individual sequences via variable–rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.