Lempel-Ziv Compressed Full-Text Self-Indexes

Diego G. Arroyuelo Billiardi
Ph.D. Student, Departamento de Ciencias de la Computación
Universidad de Chile
darroyue@dcc.uchile.cl

Advisor: Gonzalo Navarro

Outline

- 1. Introduction and Previous Work
- 2. Lempel-Ziv Compression
- 3. The LZ-index Data Structure
- 4. Our Research Topics
  - 4.1. Space-efficient Construction of LZ-index
  - 4.2. Reducing the Space Requirement of LZ-index
  - 4.3. A Secondary Memory Prototype of LZ-index
  - 4.4. A Dynamic LZ-index
- 5. Conclusions

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.
- Two approaches for solving the problem: *sequential* and *indexed* text searching.
- In our work we focus on indexed text searching.

1. Introduction and Previous Work (cont.)

- From about 1996, there are some works [13, 14, 12, 9, 28, 29, 5, 6, 7, 8, 25, 26, 27, 17, 18] presenting ***compressed indexes***, taking advantage of the regularities of the text to **operate in space proportional to that of the compressed text**.
- In many of those works 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.

1. Introduction and Previous Work (cont.)

- A concept related to text compression is that of the k-th *order empirical entropy of a text* T, denoted by $H_k(T)$ [20].
- 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.
- It holds that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$.
- **The size of the uncompressed text is**: $u \log \sigma$ bits.
- **The size of the compressed text is**: $uH_k(T)$ bits.

1. Introduction and Previous Work (cont.)

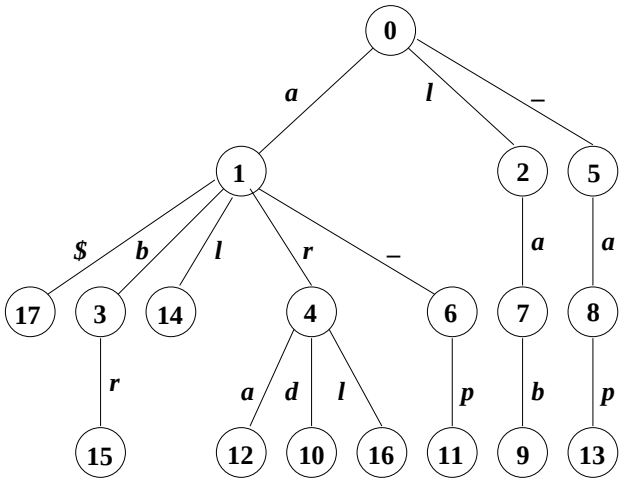
- As with text compression, handling compressed indexes increases processing time.
- But it is preferable to handle compressed indexes entirely in main memory, rather than handling them in uncompressed form but in secondary storage.
- We are interested in compressed indexes based on the Lempel-Ziv compression algorithm.
- An important property is that, if the Lempel-Ziv parsing cuts the text into n *phrases*, then $n \log u = uH_k(T) + o(kn \log \sigma)$ for any k [15].

1. Introduction and Previous Work (cont.)

- The most basic problems for compressed self-indexes are that of searching and reproducing any part of the text.
- However, a self-index must provide many other functionalities in order to be fully useful.
- In our work we propose a deep study of compressed full-text self-indexes based on the Lempel-Ziv compression algorithm.
- Specifically, we focus our studies on Navarro's LZ-index [25, 26, 27].
- We aim at a compressed full-text self-index with many interesting properties: (fast full-text searching, fast text recovery, using little space for construction and operation, allowing insertion and deletion of text, and efficient construction and search in secondary memory).

2. Lempel-Ziv Compression

- The general idea is to replace substrings in the text by a pointer to a previous occurrence of them.
- The Lempel-Ziv compression algorithm of 1978 (LZ78 [30]) 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.
- If we assume that a prefix $T_{1...j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \dots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1...u}$ 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. $Z = b_1 \dots b_r b_{r+1}$, and add the block to the dictionary.



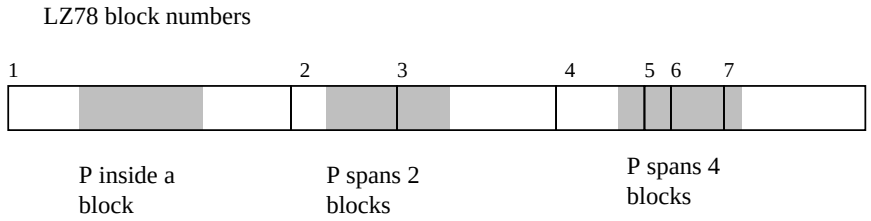
alabar a la alabarda para apalabrarla

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a	*l*	*ab*	*ar*	*_*	*a_*	*la*	*_a*	*lab*	*ard*	*a_p*	*ara*	*_ap*	*al*	*abr*	*arl*	*a$*

Properties: the dictionary is prefix-closed and a natural way to represent it is a trie. Also, every block represents a different text substring.

3. The LZ-index Data Structure

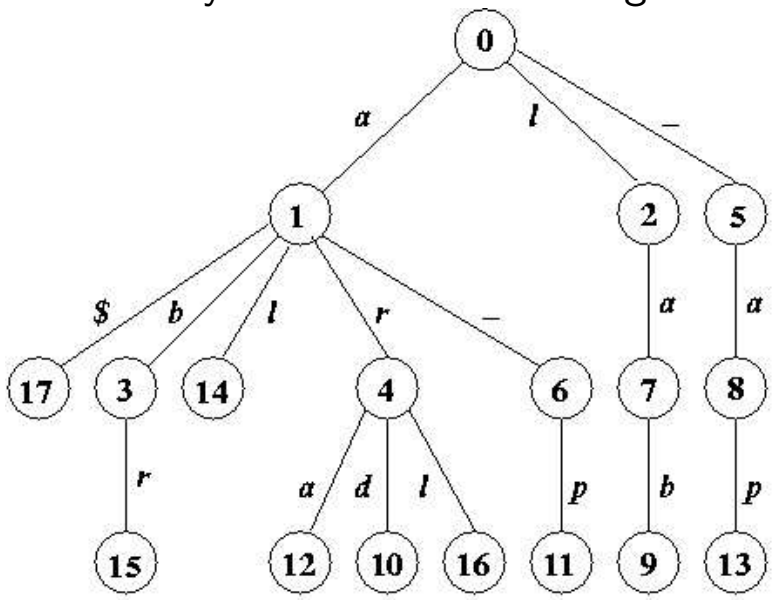
- The Navarro's LZ-index [25, 26, 27] is a compressed full-text self-index based on LZ78.
- We distinguish three types of occurrences of P in T:



- LZ-index consists of four main components (data structures), which are used to find each type of occurrence:
  - *LZTrie*: the trie formed by the LZ78 blocks $B_0, \ldots, B_n$. It has $n+1$ nodes.
  - *RevTrie*: the trie formed by the reversed blocks $B_0^r, \ldots, B_n^r$. It has empty nodes.
  - *Node* and *RNode* mappings.
- Each of these data structures requires $n \log n$ bits $= uH_k(T) + o(u)$ bits.
- The space requirement of LZ-index is $4uH_k(T)(1 + o(u))$ bits, and query time is, in the worst case, $O(m^3 \log \sigma + (m + occ) \log n)$.

3. The LZ-index Data Structure (cont.)

- The tries are represented using the *parentheses representation* of Munro and Raman [23].
- But they are constructed using a non-space-efficient representation.



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

4. Our Research Topics

In our work we propose a deep study of compressed full-text self-indexes based on the Lempel-Ziv compression algorithm.

Next we present the main parts of our work.

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.
- Construction of *compressed suffix array* (*CS-array*) [28] and *FM-index* [5] involves building first the suffix array of the text.
- Similarly, the LZ-index is constructed over a non-space-efficient 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.
- The Human Genome (≈ 3GB) may fit in 1GB of main memory using these indexes (and thus it can be operated entirely in RAM on a desktop computer), but 15GB of main memory are needed to build them!

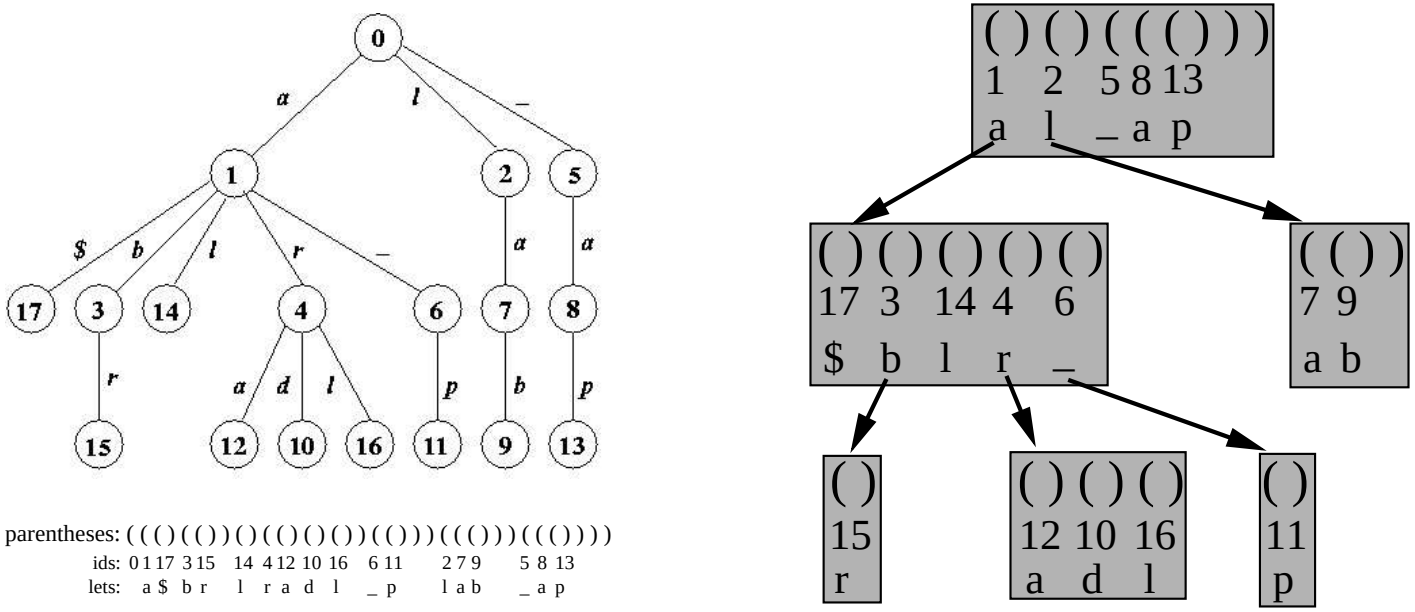
4.1. Space-efficient Construction of LZ-index (cont.)

- The works of T.-W. Lam et al. [16] and W.-K.Hon et al. [10] deal with the space (and time) efficient construction of *CS-array*.
- The main memory requirement to build the LZ-index comes from the normal tries used to build *LZTrie* and *RevTrie*.
- We aim at a practical and efficient algorithm to build those tries in little memory.
- The idea is to replace the normal tries with space-efficient **intermediate** data structures that support insertions.
- These can be seen as hybrids between normal tries and the final parentheses representations.

4.1. Space-efficient Construction of LZ-index (cont.)

- We modify the representation of balanced parentheses [23] 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.
- 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* (subsets of trie nodes), and arrange these pages in a tree of pages (the entire trie is a tree of pages).
- A page is represented as a contiguous block of memory.
- When we insert a new node in the hrbp (corresponding to a Lempel-Ziv block), we only need to recompute the page were the insertion is done.

4.1. Space-efficient Construction of LZ-index (cont.)



To achieve a minimum fill ratio α in the pages of the hrbp we define the following lemma.

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

4.1. Space-efficient Construction of LZ-index (cont.)

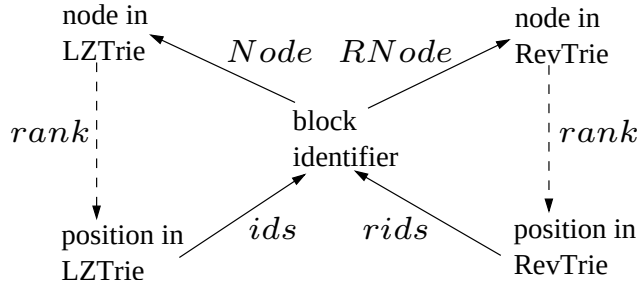
- To save space, we do not store navigation information in the pages ($O(N_t)$ time).
- Solving *page overflows* (i.e., insertion in a full page p):
  - For pages of size N_i, $1 \leqslant i \leqslant t-1$, we allocate a page q of size N_{i+1}, and copy the content of p to q (thus, the fill-ratio of q is at least α).
  - For pages of size N_t, we select a subset of nodes to be copied to a new child page.
- For the selection of subtrees, we choose the subtree of maximum size not exceeding $N_t/2$ nodes, and thus the size of the new leaf page is at least is $N_t/2\sigma$ nodes.
- In this way we minimize the number of pointer between pages, since in the worst case there are $n \log u(2\sigma/N_t)$ pointers.
- We represent *RevTrie* using a *PATRICIA tree* [21], and we ensure $n' \leqslant 2n$ nodes.
- To reduce the maximum space usage of the algorithm, we change the order in which the data structures are built: *LZTrie*, *RevTrie*, *Node*, and then *RNode*.

4.1. Space-efficient Construction of LZ-index (cont.)

- Overall, our space-efficient algorithm to construct LZ-index requires $(4+\epsilon)uH_k(T)+o(u)$ bits, and $O(\sigma u)$ time, for any constant $0<\epsilon<1$ and $k=o(\log n/\log^3\sigma)$.
- This is the first construction algorithm of a compressed full-text self-index whose space requirement is related to H_k rather than to H_0.
- In practice, the indexing space varies from 1.46 ($\alpha=0.95$) to 1.49 ($\alpha=0.5$) times the text size (English text). For $\alpha=0.95$, the maximum indexing space is reached at last step of construction (when we get the final LZ-index).
- The whole indexing rate (using a 2GHz machine) varies from 4.60sec/MB ($\alpha=0.95$) to 4.29sec/MB ($\alpha=0.5$), which is 6 times slower than original construction.
- **Main Result in Practice**: wherever the LZ-index can be used, we can build it.
- All these results have been submitted (**and accepted!** [1]) to *ISAAC 2005*.

4.2. Reducing the Space Requirement of LZ-index

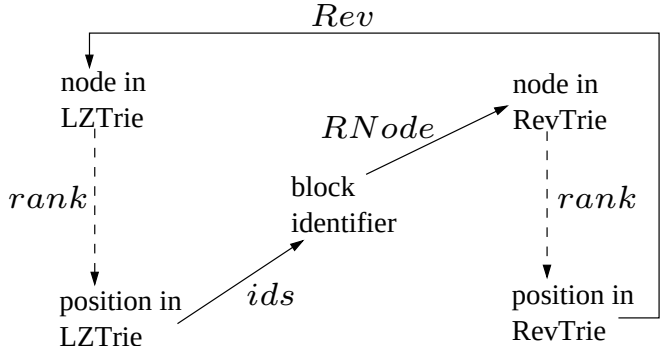
- The space requirement of LZ-index (1.2 to 1.6 times the text size) is relatively large compared with competing schemes, such as *CS-array* (0.6 to 0.7 times the text size) and *FM-index* (0.3 to 0.8 times the text size).
- LZ-index is actually a "navigation" scheme that permits us moving back and forth 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.



- The structure, however, is redundant (the number of arrows is not minimal).

4.2. Reducing the Space Requirement of LZ-index (cont.)

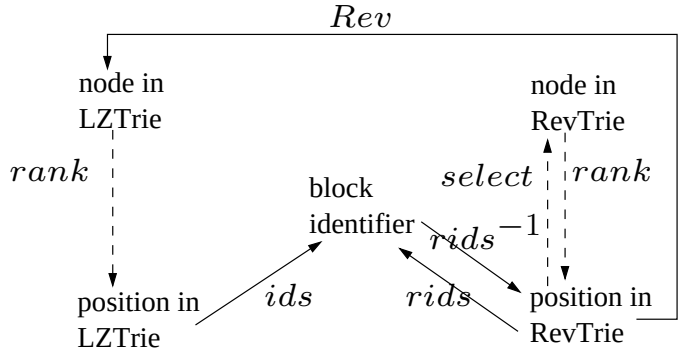
- An alternative navigation scheme is given in [24] where the minimum number of links are used.



- Arrays *rids* and *Node* have disappeared and have been replaced by mapping *Rev*.
- 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 ($Rev(rank(RNode(id)))$ vs. $Node(id)$).
- But in some cases queries are even faster under this scheme ($Rev(rpos)$ vs. $Node(rth_r(rpos))$).

4.2. Reducing the Space Requirement of LZ-index (cont.)

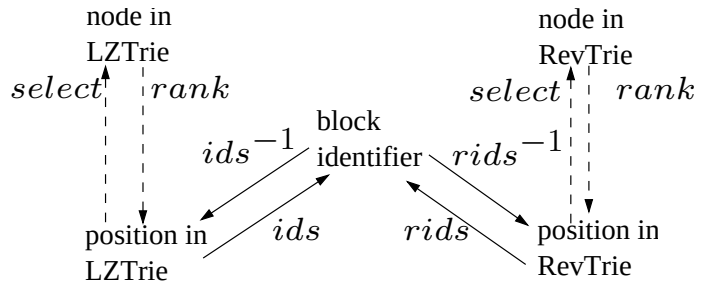
- We propose new alternatives to reduce the redundant information in the LZ-index.



- In this new scheme we replace *RNode* by $rids^{-1}$, the inverse of permutation $rids$.
- We use the idea of Munro et al. [22] requiring $(1+\epsilon)n \log n + O(1)$ bits to represent a permutation and compute its inverse in $O(1/\epsilon)$ time, for a constant $0 < \epsilon < 1$.
- We hope a space requirement of about 3/4 the space of the original LZ-index.
- **Advantage**: the largest useful path length is 2 (but paths including $rids^{-1}$ have an aditional cost).

4.2. Reducing the Space Requirement of LZ-index (cont.)

- Another navigation scheme is the following:



- We replace the Rev array by the computation of $select$ (the inverse of $rank$).
- Thus, this version of the index requires less space (we hope about 0.6 times the size of the original LZ-index).
- As the data structure of Munro et al. allows space/time trade-offs, we expect that this introduce space/time trade-offs to LZ-index.

4.3. A Secondary Memory Prototype of LZ-index

- Compressed full-text self-indexes require little memory, but 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.

4.3. A Secondary Memory Prototype of LZ-index (cont.)

- There do not exist many works on full-text indexes on secondary memory, which definitely is an important issue: the *String B-tree* [4], the *Compact Pat Trees* [3], and the *CS-array* on secondary storage [19].
- In this part of the work 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 to construct LZ-index can be useful, since it cuts the trie into pages which can be stored on secondary memory.
- The main aspect in the research will be the reduction of the navigation among index components (both at construction and search time).
- As a result, we hope to obtain an efficient version of LZ-index working on secondary memory (both for constructing and searching), making use of compression to reduce the number of disk accesses.

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.
- 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 [4, 5, 11, 2], of which the last three are compressed self-indexes. Yet, those are very preliminary.

4.4. A Dynamic LZ-index (cont.)

- 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.

- Note that the intermediate hrbp of the tries 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.
- As a result, we hope 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 Lempel-Ziv schemes.

Conclusions

- Our main goal is to allow fast full-text searching using little space.
- We study Lempel-Ziv compressed full-text self-indexes.
- 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.

References

[1] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. ISAAC'05*, LNCS. Springer, 2005. To appear.

[2] 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.

[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 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.

[5] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc FOCS'00*, pages 390–398, 2000.

[6] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pages 269–278, 2001.

[7] P. Ferragina and G. Manzini. On compressing and indexing data. Technical Report TR-02-01, Dipartamento di Informatica, Univ. of Pisa, 2002.

[8] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850. SIAM, 2003.

[9] 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.

[10] 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.

[11] 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.

[12] J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Dept. of Computer Science, University of Helsinki, Finland, 1999.

[13] 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.

[14] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, LNCS 1090, pages 219–230, 1996.

[15] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[16] 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.

[17] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, LNCS 3109, pages 420–433, 2004.

[18] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, LNCS 3537, pages 45–56, 2005.

[19] 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.

[20] G. Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[21] D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[22] I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *ICALP*, LNCS 2719, pages 345–356, 2003.

[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. Implementing the LZ-index: Theory versus practice. Unpublished personal note.

[25] G. Navarro. Indexing text using the Ziv-Lempel trie. In *Proc. SPIRE'04*, LNCS 2476, pages 325–336, 2002.

[26] 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`.

[27] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

[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] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. SODA'02*, pages 225–232, 2002.

[30] J. Ziv and A. Lempel. Compression of individual sequences via variable–rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.