Improving Text Indexes Using Compressed Permutations

Jérémy Barbay, Carlos Bedregal[1], Gonzalo Navarro[2]
Department of Computer Science
University of Chile, Chile
{*jbarbay,cbedrega,gnavarro*}*@dcc.uchile.cl*

***Abstract*—Any sorting algorithm in the comparison model defines an encoding scheme for permutations. As adaptive sorting algorithms perform $o(n \lg n)$ comparisons on restricted classes of permutations, each defines one or more compression schemes for permutations. In the case of the compression schemes inspired by Adaptive Merge Sort, a small amount of additional data allows to support in good time the access and reversed access to the compressed permutation, without decompressing it. In this paper we explore the application of two of these compressed succinct data-structures to the encoding of inverted lists and of suffix arrays, and show experimentally that they yield a practical self-index on practical data-sets, from natural language to biological data.**

I. Introduction

Building a text index is nowadays the best alternative to work with large texts. These indexes are structures built on top of the text that allow fast access and efficient search for patterns in exchange for some extra space. Even if we are able to store a large text in main memory, it is likely that we use secondary memory to store the index, which is a real problem as we want to perform operations over the text efficiently. Compression techniques take advantage of regularities in the text to build compressed text indexes, allowing efficient queries over the text and requiring space proportional to the compressed text. The study of Navarro and Mäkinen [1] covered the use of compact data structures in new compressed indexes, called self-indexes, which contain enough information to reproduce any portion of the text without accessing the original text.

Additionally, Barbay and Navarro [2] proposed compression schemes for permutations achieving better compression when certain specificities of the text arise. In this paper we evaluate the practical application of these compressed representations of permutations in the encoding of text indexes (such as inverted lists and suffix arrays) for different kinds of texts.

The paper is organized as follows. Section II summarizes the previous work done in sorting and representing permutations. Section III describes how these techniques can be applied as compression schemes for text indexes. Section IV presents our empirical results. Finally, Section V presents the conclusions and future work.

[1]Supported by Conicyt Grant.
[2]Funded in part by Fondecyt Grant 1-080019.

II. Compressed Representations of Permutations

A permutation π of the integers $[1..n] = \{1, \ldots, n\}$ can be trivially represented in $n\lceil \lg n \rceil$ bits, within $O(n)$ bits of the information theory lower bound of $\lg(n!)$ bits. The latter yields a lower bound of $\Omega(n \lg n)$ comparisons to sort a permutation in the comparison model. If we note the results of each comparison performed by a sorting algorithm, this sequence will uniquely identify the permutation sorted and therefore encode it. Adaptive sorting algorithms [3] take advantage of specificities of the permutation to sort, which make them preferable since at the cost of losing a constant factor on "bad" classes of permutations, they achieve $o(n \lg n)$ comparisons on many others.

Some applications require an efficient access to both the permutation π and to its inverse π^{-1}. If we support these operations over the compressed representation of the permutation (i.e., without having to decompress it), we can improve the functionality of previous approaches for applications such as text compression.

Estivill-Castro and Wood [3] list previous studies that focused on the effect of presortedness in sorting and how to measure this difficulty. Each of these adaptive algorithms yields a compression scheme for permutations, but the encoding defined does not necessarily support the operations $\pi()$ and $\pi^{-1}()$ efficiently.

The techniques proposed by Barbay and Navarro [2] take advantage of ordered subsequences in the permutation to produce a compressed representation. For a sorting algorithm such as merge sort, it is possible to speed up the performance of the algorithm by linearly partitioning the array into already sorted sub-arrays and later merge them in linear time [4]. The best order for merging the sub-arrays is obtained by the execution of Huffman's coding algorithm [5] over the sequence of lengths of the sub-arrays. In order to maintain the distribution of the elements of the original array, an alphabetic coding such as Hu-Tucker algorithm [6] can be used instead.

The measure of the *entropy* of a sequence of positive integers $X = \langle n_1, n_2, \ldots, n_r \rangle$ adding up to n is given by $\mathcal{H}(X) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$, which by convexity of the logarithm satisfies the property $\frac{r \log n}{n} \leq \mathcal{H}(X) \leq \log r$.

Consider a *run* in a permutation π as a maximal range of consecutive positions $[i..j]$ which does not contain any *down step* (i.e., a position p such that $\pi(p+1) < \pi(p)$). There is an encoding scheme for permutations that uses at most $n(2+\mathcal{H}(LRuns))(1+o(1))+O(\rho \lg n)$ bits to encode a permutation of size n covered by ρ runs of lengths $LRuns$ and support $\pi(i)$ and $\pi^{-1}(i)$ in time $O(1+\lg \rho)$ for any $i \in [1..n]$, or in time $O(1+\mathcal{H}(LRuns))$ for i chosen uniformly at random in $[1..n]$ [2, Theorem 3.2].

In a stricter variant of the runs, a *strict run* is defined as a maximal range of positions satisfying $\pi(i+k) = \pi(i)+k$ and the *head* of such runs is its first position. Strict runs allow further compression when they arise. There is an encoding scheme for permutations using at most $\tau\mathcal{H}(LHRuns)(1+o(1)) + 2\tau \lg \frac{n}{\tau} + o(n) + O(\tau + \rho \lg \tau)$ bits to encode a permutation of size n covered by τ strict runs and $\rho \leq \tau$ runs, where $LHRuns$ is the vector with the ρ run lengths in the permutation of strict run heads. It supports $\pi(i)$ and $\pi^{-1}(i)$ in time $O(1+\lg \rho)$ for any $i \in [1..n]$, or in time $O(1+\mathcal{H}(LHRuns))$ for i chosen uniformly at random in $[1..n]$ [2, Theorem 3.6].

In the next section we show how both compression schemes can be applied to text indexes.

III. Application in Text Indexes

A text index built over the text allows fast access and substring searching, at the cost of some additional space. Nowadays this is the best alternative for large texts, as otherwise it would require sequential traversals of the whole text. The support of operations such as search, count or locate of a given pattern allows the implementation of more complex functions; therefore efficient indexes for this queries are desirable.

Inverted indexes are very popular for text retrieval in natural language [7]. We consider a text $T[1, n]$ of n words, and ρ the number of distinct words in T (i.e., the vocabulary size). Since the concatenation of the ρ inverted lists can be seen as a permutation of $[1..n]$ with ρ runs, it can be compressed using the schemes reviewed in Section II. The resulting index can be considered a self-index as the compressed index is capable of reproducing the original text.

On the other hand, when a text cannot be handled with inverted indexes, suffix arrays are used for indexing. Consider a text $T[1, n]$ of n symbols and alphabet of size ρ. The *suffix array* $A[1, n]$ is defined as a permutation of $[1..n]$ so that $T[A[i], n]$ is lexicographically smaller than $T[A[i+1], n]$, i.e., all suffix are lexicographically ordered. Various compressed representations of suffix array were proposed since the space requirement of the uncompressed index would be high. The *Compressed Suffix Array (CSA)* of Sadakane [8] builds over a permutation Ψ of $[1..n]$, where $\Psi(i)$ stores the position in A of the next symbol of suffix $A[i]$. This permutation let us navigate one position forward in the text. Similarly, the family of FM-index [9], [10] works with an approach that allows a backward navigation of the suffixes.

IV. Experimental Results

We test two compressed representations for permutations: *runs* (Runs) and *strict runs* (SRuns). Both techniques were applied in two distinct scenarios: inverted indexes and suffix arrays. Experiments were executed on a 2 GHz Intel Xeon with 16 GB of main memory and running Ubuntu GNU/Linux. The compiler used was gcc version 4.2.4. Time results were measured in CPU user time.

A. Suffix Arrays

For general texts, we compared the proposed indexes Runs and SRuns with existing techniques for compression of suffix arrays: Compressed Suffix Array (CSA) [8], Succint Suffix Array (SSA) [10], Practical Succint Suffix Array (FSSA) [11], Run-Length FM-Index (RLFMI) [12] and the Alphabet-Friendly FM-Index (AFFMI) [10]. Four text collections were used for the experiments: *dna* (DNA sequences), *proteins* (proteins sequences), *sources* (source program code) and *xml* (structured text). The text files (all of 200 MB) were obtained from the *Pizza&Chili* repository [13].

Three configurations were used for the different indexes, corresponding to space-time tradeoffs for each technique. For CSA, the sampling of array Ψ (S_Ψ) was fixed to 128, while the sampling of the suffix array (S_A) used parameters $\{16, 32, 64\}$. For SSA, the sampling of the text (S_T) was fixed to 64 and S_A used parameters $\{32, 64, 128\}$. FSSA, RLFMI and AFFMI used sampling parameters $\{32, 64, 128\}$.

Tables I and II summarize the statistics about the ascending subsequences found in the permutation Ψ of each text. For runs, the second column of Table I shows the total number of runs found in Ψ, the third column shows the entropy of the distribution of the lengths of the runs ($LRuns$), the fourth column shows the maximum length of the runs, and the fifth shows the percentage of the permutation covered by a single run on average. For strict runs, the second column of Table II shows the total number of strict runs found in Ψ, the third column shows the entropy of the distribution of the run lengths in the permutation of strict run heads, the fourth column shows the maximum length of the strict runs in Ψ, and the fifth column shows the average length of the strict runs since the average percentage of coverage was negligible compared to the size of the text (around 10^{-6}).

Tables I and II explains the behavior of the proposed indexes for different kinds of texts, and how the distribution of runs and strict runs affects the final compression. For the four scenarios the entropy values of $LRuns$ and $LHRuns$ indicate that the strategy used for merging the runs performed better than a balanced merge algorithm, especially for the permutations of the *dna* and *sources* texts (as the entropy was inferior than $\lg \rho$). For the *sources* and *xml* texts, SRuns index achieved better compression because the strict runs

Text	# runs	$\mathcal{H}(LRuns)$	Max. run length	Avg. run coverage
dna	17	1.97	62,457,518	5.88%
proteins	26	4.20	21,534,302	3.85%
sources	231	5.47	30,323,091	0.43%
xml	97	5.26	14,302,844	1.03%

Table I
STATISTICS OF RUNS IN PERMUTATION Ψ OF THE TEXTS.

Text	# strict runs	$\mathcal{H}(LHRuns)$	Max. strict run length	Avg. run length
dna	128,863,384	1.99	675	1.6
proteins	108,458,913	4.21	9,008	1.9
sources	47,650,638	5.74	274,529	4.4
xml	29,584,818	5.53	2,195,799	7.1

Table II
STATISTICS OF STRICT RUNS IN PERMUTATION Ψ OF THE TEXTS.

tend to be longer in comparison to the strict runs found in the *dna* and *proteins* permutations. Working with runs, the *dna* and *proteins* permutations were covered by few longer runs, a favorable scenario for compression using the Runs index. On the other hand, the *sources* and *xml* permutations presented relatively short runs, and although *sources* had more than twice the number of runs of *xml*, compression ratios were similar due to their close values of $\mathcal{H}(LRuns)$. Table III summarizes the memory usage of Runs and SRuns indexes.

Figure 1 shows the space-time tradeoffs for evaluating Ψ. We measured the average time (in microseconds) of accessing Ψ at 100,000 random positions. In this scenario we compared the compression techniques based on runs (Runs) and strict runs (SRuns) to Sadakane's CSA, as this index compresses the suffix array via the function Ψ that captures text regularities and allows forward navigation inside the text. As shown in Figure 1, CSA's times are smaller than Runs and SRuns indexes in every scenario (this could be due to the fact that CSA also takes advantage of the ascending runs present in Ψ).

The distribution of ascending subsequences (runs and strict runs) in each text is reflected in the different – but competitive– ratios of compression. Although relatively short, the presence of strict runs in the texts *proteins* and *xml* let SRuns index achieve better compression than Runs,

Text	Runs	SRuns
dna	0.42	0.52
proteins	0.58	0.57
sources	0.74	0.44
xml	0.71	0.37

Table III
MEMORY USAGE OF RUNS AND SRUNS (FRACTION OF TEXT).

Text	size (bytes)	num. words	voc. size
english	1,073,741,813	238,781,975	622,834

Table IV
DESCRIPTION OF THE TEXT USED FOR NATURAL LANGUAGE.

with comparable times for evaluating Ψ. For the texts *dna* and *proteins*, where typical runs are more common, the space requirement of the Runs index is lower than the one required by CSA. Even when CSA performs better in time, Runs and SRuns indexes do not depend as much on sampling parameters as CSA does (S_Ψ could be modified to reduce the space, but this would negatively affect the access time to Ψ). In contrast to CSA, both Runs and SRuns behave as a bidirectional index since they allow both forward and backward navigation inside the text.

Figure 2 shows the space-time tradeoffs for evaluating Ψ^{-1}. We measured the average time required to evaluate Ψ^{-1} at 100,000 random positions of the text. In this scenario we compared the compression techniques based on runs (Runs) and strict runs (SRuns) to the group of indexes from the FM-index family [9], such as Succint Suffix Array (SSA), Practical Succint Suffix Array (FSSA), Run-Length FM-Index (RLFMI) and the Alphabet-Friendly FM-Index (AFFMI), since these indexes are built using the BWT and backward searching, allowing backward navigation inside the text.

Besides taking advantage of the presence of runs and strict runs, in general our indexes performed better in terms of time and space. Within a lower space requirement, Runs and SRuns indexes achieved faster times calculating Ψ^{-1}. The same observations about the runs distribution can be noted in this scenario (indexes Runs and SRuns are the same as in the previous experiment). Figures 1 and 2 illustrated the superiority of Runs and SRuns indexes for bidireccional navigation inside the text, a feature that can be used, for example, in operations that required random access to the text or extraction of snippets of variable lengths (lines, paragraphs, etc.).

B. Inverted Indexes

For natural language, we applied the compression techniques based on runs (Runs) and strict runs (SRuns) in inverted indexes and compared them to WPH [14], a competitive text index that improves over the Plain Huffman coder [15]. The *english* text collection contains the concatenation of English texts selected from *etext02–etext05* of the Gutenberg Project. The file was obtained from the *Pizza&Chili* repository [13]. Table IV shows some statistics of the text.

Table V shows the compression ratio obtained by each technique. Runs represents the compression using ascending runs while SRuns represents the compression using strict runs as seen in Section II. The amount of memory usage of

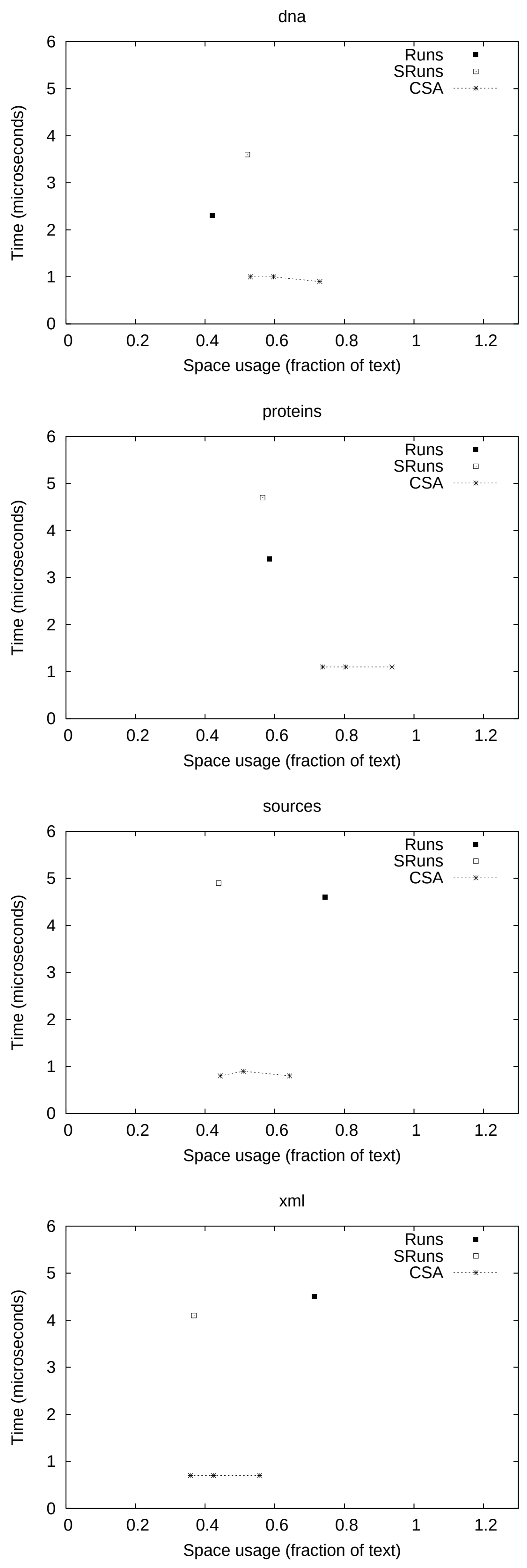

Figure 1. Space-time tradeoffs for evaluating Ψ.

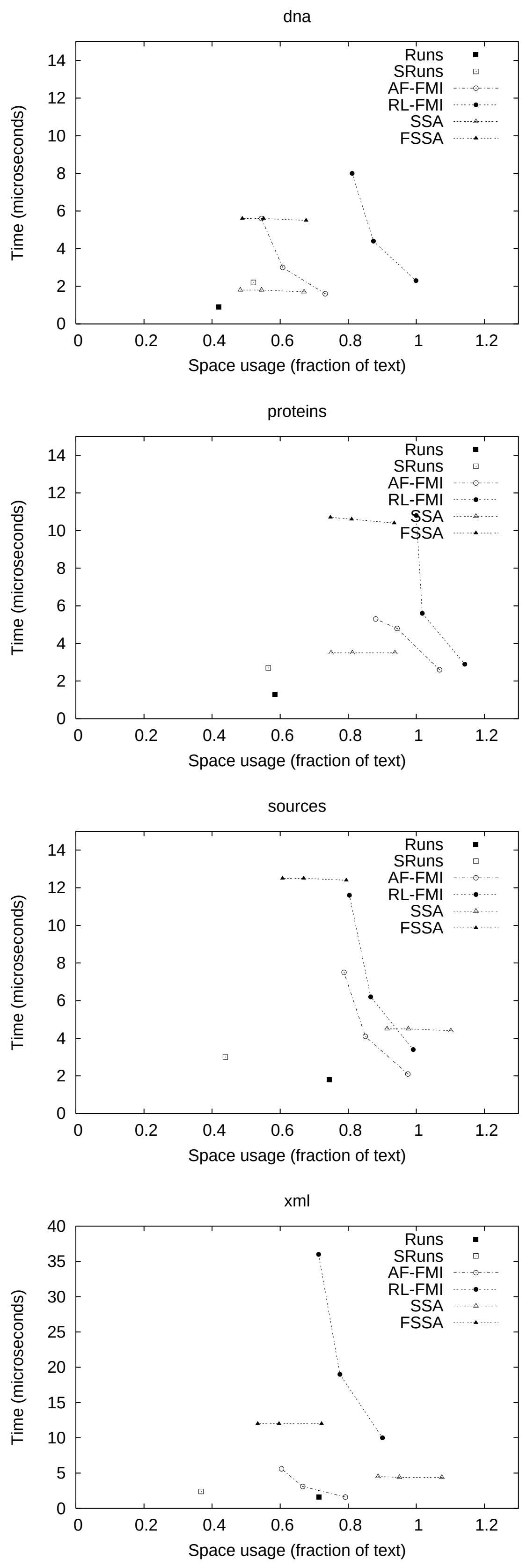

Figure 2. Space-time tradeoffs for evaluating Ψ^{-1} (LF).

Text	Runs	SRuns	WPH
english	0.32	0.38	0.36

Table V
MEMORY USAGE OF EACH INDEX (FRACTION OF THE ORIGINAL TEXT).

Query	Freq.	Runs	SRuns	WPH
Locate	1-100	0.00005	0.00004	0.00008
	101-1000	0.00171	0.00190	0.00813
	1001-10000	0.01260	0.01381	0.01583
	>10000	0.11140	0.12830	0.09676
Snippet	1-100	0.00010	0.00015	0.00046
	101-1000	0.00305	0.00486	0.02303
	1001-10000	0.03110	0.03970	0.12536
	>10000	0.44850	0.56300	1.36786

Table VI
PERFORMANCE OF THE INDEXES FOR DIFFERENT WORD FREQUENCIES (TIMES IN SECONDS).

the Runs and SRuns encodings are similar to that required by WPH; although Runs achieves a better compression, SRuns does not achieve a good ratio because of the lack of strict runs in the permutation (in this case, a strict run in the permutation comes from consecutive words in the text that are lexicographically one after another). Statistical measures on the text showed that the average run size is 511 while the average strict run size is 1; this explains how the presence –or absence– of runs in the text directly affects the final compression obtained.

Table VI shows the performance of the indexes when searching for words. We compare the time to *locate* all the text occurrences of a pattern and the time to extract all the *snippets* around each of these occurrences. For both scenarios we consider words from 4 different ranges of frequency as shown in Table VI. We calculate the average time per pattern from 100 randomly-chosen single-word patterns. The snippets were obtained extracting a context of 10 words, starting 5 words before the occurrence.

Both operations of location and extraction of snippets are faster using our compression schemes. For the case of locate, the resulting times of the WPH index were close, especially for very frequent words, where WPH index was slightly faster. For extracting snippets, the Runs and SRuns indexes were on average 4 times faster than WPH, which is a great advantage considering that the Runs index requires less space to operate. In Runs and SRuns indexes, we obtained the snippets from the inverse permutation π^{-1}, while locate queries were done accessing π. Since the former is performed faster than the latter, operations of extraction will perform very fast for both indexes.

V. Conclusions and Future Work

In this paper we have shown how sorting algorithms can inspire techniques in data compression. Reducing the text to a permutation, it is possible to take advantage of ordered consecutive intervals and use them to improve the compression. Our indexes have proven to be competitive in terms of space when the runs arise, and in terms of time, the indexes were still competitive for some basic text operations. The bidirectional indexes obtained could allow, for example, operations that display the context around a pattern occurrence without requiring extra space. More experiments are required to exhaustively compare the performance of these indexes for more complex operations.

In general, the compressed representation of permutations is a promising technique for applications such as text compression. Adaptive sorting algorithms suggest new schemes for compression, with their measures of difficulty yielding new measures of compression.

Other adaptive algorithms, such as *Inv* (pairs of elements in the wrong order) or *Rem* (elements which have to be removed to leave the list sorted), will define new compression schemes for permutations; it is of interest to evaluate if they can support operations (i.e., access to the permutation) in reasonable time.

This work can also be extended to include indexes based on Shuffled UpSequences (SUS) and Shuffled Monotone Subsequences (SMS), which are measures of presortedness related to the ones used in this paper. Although computing the optimal distribution of SUS and SMS in a permutation is more complex, these indexes might be interesting when good distributions arise.

This research suggests the need for a deeper study of the relation between algorithms and encodings in contexts other than permutations, and how this time–space relation can be exploited to develop new simple and practical techniques for data compression.

References

[1] G. Navarro and V. Mäkinen, “Compressed full-text indexes,” *ACM Computing Surveys*, vol. 39, no. 1, p. article 2, 2007.

[2] J. Barbay and G. Navarro, “Compressed representations of permutations, and applications,” in *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*. Schloss Dagstuhl, Leibnitz Zentrum fuer Informatik, Germany, 2009, pp. 111–122.

[3] V. Estivill-Castro and D. Wood, “A survey of adaptive sorting algorithms,” *ACM Computing Surveys*, vol. 24, no. 4, pp. 441–476, 1992.

[4] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[5] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[6] T. C. Hu and A. C. Tucker, “Optimal computer search trees and variable-length alphabetical codes,” *SIAM Journal on Applied Mathematics*, vol. 21, no. 4, pp. 514–532, 1971.

[7] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.

[9] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[10] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transactions on Algorithms*, vol. 3, no. 2, p. 20, 2007.

[11] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, ser. LNCS 5280. Springer, 2008, pp. 176–187.

[12] V. Mäkinen and G. Navarro, "Succinct suffix arrays based on run-length encoding," *Nordic Journal of Computing*, vol. 12, no. 1, pp. 40–66, 2005.

[13] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed text indexes: From theory to practice," *ACM Journal of Experimental Algorithmics (JEA)*, vol. 13, p. article 12, 2009, 30 pages.

[14] N. Brisaboa, A. F. na, S. Ladra, and G. Navarro, "Reorganizing compressed text," in *Proc. 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM Press, 2008, pp. 139–146.

[15] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," *ACM Transactions on Information Systems (TOIS)*, vol. 18, no. 2, pp. 113–139, 2000.