PhD Thesis Proposal

Compressed Data Structures based on Adaptive Algorithms

Carlos E. Bedregal
cbedrega@dcc.uchile.cl
Advisors:
Gonzalo Navarro, gnavarro@dcc.uchile.cl
Jérémy Barbay, jbarbay@dcc.uchile.cl

Universidad de Chile, Chile
April, 2011

Abstract. Efficient access to large data collections is nowadays an interesting problem for many research areas and applications. A recent conception of the time-space relationship suggests a strong relation between data compression and algorithms in the comparison model. In this sense, efficient algorithms could be used to induce compressed representations of the data they process. Examples of this relationship include unbounded search algorithms and integer encodings, adaptive sorting algorithms and compressed representation of permutations, or union algorithms and encoding for bit vectors. In this thesis, we propose to study the time-space relationship on different data types. We aim to define new compression schemes and compressed data structures based on adaptive algorithms that work over these data types, and to evaluate their practicality in data compression applications.

1 Introduction

Nowadays computer applications are likely to process and manage large amounts of data: text, images, videos, biological sequences, signals, and so on. Although massive storage is easily available, the real problem is efficient access to the data. For example, CPU caches are many times faster than main memory, while this in turn is many times faster than hard drives. Given this hierarchy of memories, it is preferable to work with compressed representations of the data that may fit higher levels of memory and allow operations over the data without decompressing it. This approach has been successfully applied in large text collections through compressed full-text indexes [58].

The relation between algorithms and encodings was first suggested by Bentley and Yao [8], who showed how adaptive search algorithms in sorted arrays defined a family of adaptive prefix encodings for integers. Bentley and Yao representation for integers is given by the binary result of the sequence of comparisons performed by the algorithm to find a number in an unbounded context; the encodings obtained were isomorphic to the codes proposed by Elias [20].

Later, Barbay and Navarro [5] showed how some adaptive sorting algorithms in the comparison model yield useful compressed representations for permutations. Analogous to the approach by Bentley and Yao, the sequence of binary results of the comparisons performed by the algorithm while sorting a given permutation can uniquely identify a permutation, thus encoding it. This time-space relationship can be extended to other algorithms in the comparison model working over other encodable objects, e.g. algorithms for merging sets that encode bit-vectors [1].

Considering these previous results, there exist algorithms in the comparison model that can generate a representation of the data they process; for compression purposes the obtained representation must uniquely identify the instance. Since fewer comparisons represent fewer bits to store in the sequence of binary results delivered by the algorithm, faster algorithms on some classes of instances yield shorter representations of those instances. If we consider that – in practice – some instances of the data are easier to process by an algorithm than others, adapting to the instance they face, an *adaptive* algorithm takes advantage of certain specificities present in the instance to perform faster on those instances, defining a compressed representation of the processed data. Although these representations are useful for data compression, the interesting problem is to obtain an index data structure based on the compressed representation, allowing operations over the data without decompressing it.

In this thesis we propose to study the time-space relationship in data types such as integers, sets and permutations. We plan to consider the study of the adaptive algorithms and their relation with data compression: how to use this relationship to develop new compressed data structures with efficient support for operations, keeping a low space requirement in main memory. We also plan to consider evaluations of the practical benefits of this approach in applications such as integer compression, compressed bit-sequences and text indexes.

2 Related Work

In this section we review and discuss the related work within the scope of the time-space relationship we plan to address, specifically the developments in the areas of compact data structures and adaptive algorithms. First, we introduce previous notions of compressibility and compression heuristics. Then, we discuss the usage of compact data structures in scenarios such as sequences or text collections. This data structures have proven to be useful tools for data compression, providing at the same time fast operations over the data. Finally, we describe some known adaptive algorithms working on the comparison model for the problems of unbounded search, set union, and sorting of arrays.

2.1 Measures of Compressibility

Let S be a sequence of size n over an alphabet Σ of size $|\Sigma| = \sigma$. A trivial representation of S requires $n\lceil\log\sigma\rceil$ bits, differentiating the element of the alphabet in each position [1].

The well-known *zero-order empirical entropy* [50] measures the compressibility of a sequence of symbols considering the statistical information of the sequence. Let $n_c > 0$ be the number of occurrences of a symbol $c \in \Sigma$ inside S. The zero-order empirical entropy of S is defined as

$$\mathcal{H}_0(S) = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c}.$$

The value of $n\mathcal{H}_0(S)$ represents the output size of an ideal compressor which uses $\frac{n}{n_c}$ bits for encoding a symbol c. It also represents a lower bound for any compressor that assigns a unique fixed codeword to each symbol of the alphabet. Working with a binary alphabet ($\sigma = 2$), the zero-order entropy of a binary sequence B of size n is defined as $n\mathcal{H}_0(B) = m\log\frac{n}{m} + (n-m)\log\frac{n}{n-m} = m\log\frac{n}{m} + O(m)$, where m is the number of bits set in B.

A stronger measure, *k-th order empirical entropy* [50], achieves better compression when the codeword is chosen in function of the k preceding symbols inside the sequence. Let w_s denote the non-empty subsequence of S formed by the concatenation of the symbols following each occurrence of $w \in \Sigma^k$ in S. The k-th order empirical entropy is defined as

$$\mathcal{H}_k(S) = \sum_{w\in\Sigma^k} \frac{|w_s|}{n}\mathcal{H}_0(w_s).$$

The k-th order empirical entropy provides a lower bound to the compression we can achieve if we encode each symbol considering the context formed by k most recently seen symbols. Additionally, $n\mathcal{H}_k(S)$ provides a measure of the index size as compared to the size the best k-th order compressor would achieve. For any sequence S and $k \geq 0$, it holds that $\mathcal{H}_{k+1}(S) \leq \mathcal{H}_k(s)$.

Another measure of compressibility for sequences, the *gap measure* [34], considers the gaps between consecutive symbols in S. Let g_i be the ith gap of S, g_i is equal to the difference between the ith symbol of S and the previous one. The lower bound to encode the sequence of gaps $G = g_1, \ldots, g_n$ in binary is given by

$$gap(S) = \sum_{i=1}^{n} \lceil\log(g_i + 1)\rceil.$$

It is also possible to take advantage of the gap lengths distribution to reduce the space requirement by using prefix codes such as Elias-codes [20]. In the worst scenario, $gap(S)$ is maximized when each of the n gaps are the same (i.e. σ/n), requiring $n\log\frac{\sigma}{n}$ bits to encode G. In contrast to the empirical entropy, the gap

[1] Throughout this proposal log denotes the binary logarithm.

measure is usually less than the information-theoretic lower bound, specially for practical data sets [34].

The *trie measure* [34], another compressibility measure for sequences, is based on the *prefix omission method (POM)* [41], which represents each element of S with respect to the previous element by omitting the common prefix they share. If we consider each element of S as its $(\log \sigma)$-length bit representation, and T the trie of n leaves built using these representations, the trie measure $trie(S)$ is equal to the number of edges in T, reducing the space usage when prefixes are very common. In relation with the gap measure, Gupta *et al.* [34] proved that $trie(S) \leq 2gap(S)$.

2.2 Compact Data Structures

Motivated by the high access costs required by large data collections, research in the area of data structure has been focused in the development of *compact data structures*, aiming to reduce the space requirements to a proportion of the data representation while supporting interesting operations without uncompressing the whole data structure. Compact data structures can easily replace classical pointer-based data structures, improving the performance in both time and space as they can maintain more data in higher levels of memory. We can find a wide range of applications implementing compact data structures, from binary sequences, to compressed self-indexes for text.

Binary Sequences Consider a binary sequence as a sequence B of size n over a binary alphabet. Working with binary sequences, the *rank* and *select* operations are present in problem domains, especially in applications such as compressed full-text indexes [58]. A *rank* operation answers the number of occurrences of a symbol until a given position of the sequence, while a *select* operation, the position of a given occurrence of a symbol inside the sequence. The first study of a succinct representation to solve this operations was done by Jacobson [38], who achieved constant time for rank using a dictionary of $o(n)$ bits additional to the sequence. Later, Munro [54] and Clark [14] improved the $n + o(n)$ solution to achieve constant time for both rank and select, but they did not take advantage of the compressibility of the sequence. The work by Raman *et al.* [59] supported rank and select operations in constant time achieving compression at the same time, as this representation required $n\mathcal{H}_0(B) + o(n)$ bits. Recent studies (Mäkinen *et al.* [47], Gupta *et al.* [34]) focused on lowering the space required to $gap(B)(1 + o(1))$ of the sequence, encoding the distances between 1-bit runs inside the sequence. Practical implementations of data structures for binary sequences, as the one by Gonzales *et al.* [31], sacrifice some theoretical guarantees to improve the performance.

General Sequences For an arbitrary sequence S of size n over an alphabet of size σ, the *Wavelet Tree* [33] is a well-known data structure that decomposes the sequence into bitmaps, reducing rank/select/access operations on strings

to rank/select operations on binary sequences. It supports rank, select and access operations in $O(\log \sigma)$ time using $n\lceil \log \sigma \rceil(1 + o(1))$ bits of space. Using Raman *et al.* representation for the internal binary sequences the space requirement is reduced to $n\mathcal{H}_0(S) + o(n \log \sigma)$ bits, maintaining the same times as before. It is important to note that, through this approach, compression schemes for binary sequences are instantly translated into new compression for general sequences. Another known representation proposed by Golynski *et al.* [30] requires $n \log \sigma + no(\log \sigma)$ bits of space, and supports either 1) rank and access in $O(\log \log \sigma)$ time, and select in $O(1)$ time, or 2) rank in $O(\log \log \sigma \log \log \log \sigma)$, access in $O(1)$ time, and select in $O(\log \log \sigma)$ time. In both scenarios the space requirement is dominated by the internal representation of permutations used.

Representations for *permutations* of integers $[1..n] = \{1, \ldots, n\}$, have been also extensively studied. In this scenario we would want to support forward $(\pi())$ and inverse $(\pi^{-1}())$ access to the permutation in reasonable time. Munro *et al.* [55] proposed a representation based on back pointers inside the cycles of the permutation, computing $\pi(i)$ in constant time and $\pi^{-1}(i)$ in time $O(t)$ $\forall i \in [1..n]$. This representation uses at most $(1 + 1/t)n \log n + O(n)$ bits storing shortcuts every t steps, although it does not compress the permutation. Recently, Barbay and Navarro [5] proposed a group of representations for permutations that achieved compression taking advantage of the already-sorted subsequences inside the permutation. Considering ρ as the minimum number of sorted subsequences covering the permutation, the simplest of these representations computes $\pi(i)$ and $\pi^{-1}(i)$ in time $O(1 + \log \rho)$ within a space requirement proportional to the zero-order empirical entropy of the ρ sorted subsequences lengths.

Text indexes Text indexes are nowadays the best alternative to work with large text collections. 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; a real problem as we want to perform operations over the text efficiently. Text compression techniques take advantage of regularities in the text to build compressed text indexes, allowing efficient queries over the text in a space proportional to the compressed text. The study of Navarro and Mäkinen [58] covered the use of compact data structures in compressed indexes, called *self-indexes*, which contain enough information to reproduce any portion of the text without accessing the original text. Ferragina *et al.* [22] showed the practical relevance of these data structures in various types of texts.

One of the most popular compressed indexes, the *Compressed Suffix Arrays (CSA)*, were initially conceived as an optimization of the *suffix array* data structure [48]. Mäkinen [45] and Grossi and Vitter [33] exploited the regularities of the text to reduce the space usage of the suffix array, though the text was still maintained explicitly. Later, Sadakane [62] turned the representation of the suffix array into a self-index, making now possible to operate without storing the text, with a space usage depending on the zero-order empirical entropy of the

text. Sadakane's CSA represents the text – and the actual suffix array – through the function Ψ (which tells where in the suffix array lies the pointer following the current one). The Compressed Suffix Array by Grossi *et al.* [33], also a self-index, is based on a new representation of function Ψ, improving the space usage of previous representations so it depends on the k-th order entropy of the text.

Ferragina and Manzini [23] introduce another approach to compress the suffix arrays based on the *Burrows-Wheeler Transform (BWT)* [12] of the text. Also called *FM-Index*, it was the first index to achieve a space usage depending on the k-th order empirical entropy of the text, although its main drawback was the alphabet dependence. Further work on this problem generated a whole family of indexes, such as the Wavelet Tree FM-Index [61], the Huffman FM-Index [32], the Run-Length FM-Index [46], and the Alphabet-Friendly FM-Index [25].

An alternative approach to self-indexing text, based on the *Lempel-Ziv compression* [44, 65] rather than on trying to compress the suffix array, identifies repeated substrings inside the text and replace them with a pointer to their previous occurrence, partitioning the text into phrases. Lempel-Ziv based indexes, called *LZ-Indexes*, also achieve space usage proportional to the k-th order empirical entropy of the text, with very competitive times when extra space is allowed [24, 57].

Trees An ordinal tree is a rooted tree in which the children of a node are ordered and specified by their ranks. The various techniques for encoding ordinal trees are grouped in three main categories: balanced parentheses sequence (BP) [38,56,63], level-order unary degree sequence (LOUDS) [17,38], and depth-first unary degree sequence (DFUDS) [7]. They support different sets of operations in constant time using $2n + o(n)$ bits for a tree of n nodes and its index, a space requirement asymptotically tight to the information-theoretic lower bound.

Graphs Compact data structures have been also in applications that manage massive graphs such as the web graph and social networks. One of the most competitive approaches, the Webgraph framework [9], exploits locality and similarity to improve compression and access times, the encoding of adjacency lists is done using previous adjacency lists plus a list of differences, but this approach depends on the node ordering and parameters such as the window size. Claude and Navarro [15] approach compresses the web graph through a grammar-based compression technique known as Re-Pair [43]. Although WebGraph can achieve better compression, the Re-Pair representation provides faster navigation within equal space requirement. A recent approach by Buehrer and Chellapilla [11], considers the use of communities on the graph to represent it in a compressed form. This approach considers a community as a directed bipartite cliques, replacing them with virtual nodes to effectively reduce the number of edges in the graph.

2.3 Adaptive Algorithms

For some computational problems there are classes of instances easier to solve than others, introducing a notion of *difficulty* of the instance. For example, sort-

ing the permutation $a = \{6, 5, 4, 3, 2, 1\}$ in the comparison model could be easily done if we determine that it is already sorted in descending order; and permutation $b = \{4, 5, 6, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ could be sorted faster if we find the two already sorted blocks and switch them. Each strategy has its own way to measure the *disorder* of the input permutation, defining a classification of the instances based on their difficulty.

An *adaptive algorithm* in the comparison model determines the kind of instance that it is facing in order to take advantage of specificities of the instance in order to accelerate the processing. Adaptive algorithms can thus perform faster over some *easy* classes at the cost of losing a constant factor on more difficult classes.

Adaptive algorithms have been widely studied for computational problems such as sorting of permutations [42, 49, 51], unbounded search of integers [8], set intersection and union [18, 37] and convex hull [13, 40].

Unbounded Search of Integers Searching for the position of a key inside an ordered list is a particular interesting problem. In the comparison model, the well-known binary search algorithm is a good example of an optimal worst case algorithm. If we consider that some instances are easier than others (e.g. when the key is in the beginning of the list), we can take advantage of the distribution of the instances to improve the performance of the algorithm. The main drawback of a binary search algorithm is that its behavior is oblivious to the position of the key. Algorithms that solve the problem of *unbounded searching* [8] (where the search is done in a unbounded key space) can be used as an adaptive mechanism to solve the bounded searching problem, paying a search cost proportional to the position of the item independently of the list size. A good example of strategies for unbounded searching techniques were proposed by Bentley and Yao [8].

Union of Sets Given two sorted lists A and B, the union problem is to generate a sorted list C containing the elements in $A \cup B$, a scenario that arises in various application domains such as databases, web search or document information retrieval. A more general problem is to merge k lists, solved either by a k-merge algorithm (simultaneously merging k lists), or by $k-1$ recursive executions of a 2-way merge algorithm (merging the two smallest list in each iteration).

For a comparison-based union algorithm s, let $K_s(m, n)$ represent the number of comparisons performed by s to merge two lists of sizes m and n in the worst case (with $m < n$) then $K_s(m, n) \geq I(m, n)$, where $I(m, n) = \lceil \log \binom{m+n}{m} \rceil$ is the information-theoretic lower bound. Existing algorithms in the comparison model, such as those proposed by Hwang and Lin [37] and Demaine *et al.* [18], perform $O(m \log(\frac{n}{m} + 1))$ comparisons in the worst case, being asymptotically optimal. Both algorithms take advantage of the gaps between elements of a set (i.e. runs of same-set elements in the resulting union set) to reduce the number of comparisons needed.

Sorting Arrays The performance of sorting algorithms in the comparison mode is lower bounded by $\Omega(n \log n)$ comparisons to sort an array of size n. Adaptive sorting algorithms take advantage of some existing order in the input, performing in function of the size of the array and the disorder in it. These algorithms are preferable as nearly sorted sequences are common in practical applications, improving on existing sorting algorithms that are oblivious to the order in the input.

Known examples of adaptive sorting algorithms include *Straight Insertion Sort*, which takes advantage of `Inv`, the number of pairs in wrong order [51]; or *Merge Sort* which takes advantage of `Run`, the number of ascending subsequences. Estivill-Castro and Wood [21] list previous studies on the effect of presortedness in a comparison-based sorting algorithm. As there could many optimal algorithms depending on the difficulty measure applied, Estivill-Castro and Wood also provide a hierarchy of disorder measures.

3 Notions of the Time-Space Relationship

In this section we discuss previous findings of the time-space relationship that suggest a deeper study in the topic. We focus on the contexts of integer encodings and their relation with unbounded searching algorithms [8], compression of binary sequences using optimal merge algorithms [1], and compressed data structures for permutations inspired by adaptive sorting algorithms [4, 5].

3.1 Unbounded Searching Algorithms and Integer Encodings

Working in the comparison model, the cost associated with an unbounded searching algorithm is expressed in terms of the number of comparison it requires to locate an integer inside an unbounded key space (also known as the problem of *unbounded searching* [8]). Unbounded searching algorithms are also used to search in large ordered tables, taking advantage of the position of the input, with a cost proportional to the item's distance from the front of the table.

As introduced by Bentley and Yao [8], any algorithm for the problem unbounded searching defines an encoding scheme for integers. Furthermore, there exists an isomorphism between the algorithms they proposed and the codes studied by Elias [20].

Given the binary sequence storing the answers of the comparisons performed by a deterministic unbounded searching algorithm, it represents a prefix-free encoding of integers: a time-space relationship useful to obtain smaller encodings for integers when we use efficient search strategies.

Integer encodings have been widely used in data compression applications [64]. Besides the close relation between Elias-codes and the algorithms proposed by Bentley and Yao, it is possible to map other popular encoders for integers such as Rice-codes [60] and Golomb-codes [29] onto comparison-based algorithms that solve the unbounded searching problem. The latter relation suggests that, in fact, both problems could be the same.

Current work aims to support operations over the compressed representations of integers. Bedregal and Tellez [6] focused on the support of arithmetic operators (e.g. addition, multiplication) over Elias-codes, minimizing the number of machine words that need to be updated. Further study in this topic include the practical evaluation of this approach, which can be useful for example in scenarios where updating the integers is too expensive.

3.2 Merge Algorithms and Compression of Binary Sequences

Ávila and Laber [1] suggested that any merge algorithm in the comparison model defines an encoding scheme for bit vectors. Based on this premise, they see optimal merge algorithms as entropy coders for, and therefore we can use adaptive algorithms for merging two sorted lists to generate compression schemes for binary sequences. To better understand this time-space relation we review how this mapping works.

Given two disjoint linearly sorted sets $A = \{a_1 \dots a_m\}$ of size m, and $B = \{b_1 \dots b_n\}$ of size n, with $m \leq n$, the merging problem consists of determining the linear ordering of their union. Considering a $compare(a_i, b_j)$ function that compares an element a_i from set A and an element b_j from set B, a comparison-based merge algorithm performs a sequence of *compare* operations in order to merge A and B.

Then, let S be a binary sequence of size n and $S(i)$ the i-th symbol of S. We define the set $A^S = \{i|S(i) = 0\}$ as the positions of the 0s in S, and $B^S = \{i|S(i) = 1\}$ as the positions of the 1s in S, with $n_0 = |A|$ the number of bits 0 in S and $n_1 = |B|$ the number of bits 1. The binary sequence R with the results of the comparisons performed by an algorithm merging the two disjoint linearly-ordered sets A^S and B^S represents an encoding of the original string S – as we can recreate S using the information in R.

For example consider $S = (1001)$, then $A^S = 2, 3$ and $B^S = 1, 4$. The output sequence of the *compare* operations performed by the tape merge algorithm to merge A^S and B^S is $>, <, <$; then $R = (100)$. To recover the sequence S from R we construct two ordered sets $A' = \{a_1, \dots, a_{n_0}\}$ and $B' = \{b_1, \dots, b_{n_1}\}$, and apply the merge algorithm over A' and B'. Although the values in A' and B' are not known, we can do the merge because the results of the comparisons performed are stored in R. Finally, we can use the obtained ordered set $C = \{c_1, \dots, c_{n_0+n_1}\}$ to reconstruct S by setting $S(i) = 0$ if $c_i \in A'$ or $S(i) = 1$ if $c_i \in B'$.

For any asymptotically optimal merging algorithms, the number of comparisons performed to merge sets A' and B' is $O(I(n_0, n_1))$ in the worst case (as seen in Section 2.3). Considering that $I(n_0, n_1) \in O(n\mathcal{H}_0(S))$, we can use optimal algorithms in the comparison model as binary entropy coders.

Examples of the time-space relation between merge algorithms and encodings for binary sequences found by Ávila and Laber [1] include: a runlength-based representation using Rice-codes [60] for same-bit runs which is related to the Binary Merging algorithm [37], or a representation similar to the Binary Interpolative Coder [52] which is related to the Recursive Merging algorithm [19].

Although the algorithm-encoding relation shown is solid, none of these representations support operations such as, access, rank or select over the binary sequence.

For a binary sequence B of size n, well-known representations already achieve constant time for rank and select operations using $n + o(n)$ bits [14, 54] or $n\mathcal{H}_0(B) + o(n)$ bits of space [59]. Other studies considered the *gap measure* to build data-aware representations for binary sequences [28,34,47,62], although they were done independently of the studies on adaptive merge algorithm using similar measures [18, 37].

3.3 Adaptive Sorting Algorithms and Compression of Permutations

On the one hand, a permutation π of the integers $[1..n] = \{1, \ldots, n\}$ can be trivially represented in $n\lceil \log n \rceil$ bits, within $O(n)$ bits of the information-theoretic lower bound of $\log(n!)$ bits. The latter yields a lower bound of $\Omega(n \log n)$ comparisons to sort a permutation in the comparison model. On the other hand, adaptive sorting algorithms [21] take advantage of specificities of the permutation to sort, achieving $o(n \log n)$ comparisons on many easy classes of permutations at the cost of losing a constant factor on more difficult classes.

Barbay and Navarro [5] observed that the binary sequence of results of the comparisons performed by a sorting algorithm in the comparison model also describes an encoding of the permutation, as this sequence will uniquely identify the permutation. They showed that adaptive sorting algorithms in the comparison model yield compression schemes for permutations, and with some extra work a compressed data structure supporting access to the permutation, $\pi()$, and its inverse, $\pi^{-1}()$.

The techniques proposed by Barbay and Navarro [5] take advantage of sorted subsequences in the permutation to produce a compressed representation. For a merge sort algorithm it is possible to speed up the performance of the algorithm by linearly partitioning the array into already sorted subsequences to later merge them in linear time [42]. The merging cost is then reduced rebalancing the merging tree via the execution of the Huffman coding algorithm [36] over the sequence of lengths of the identified sorted subsequences. In order to maintain the initial distribution of elements in the array, it is preferable to use an alphabetic coding such as the Hu-Tucker algorithm [35] instead. The sorting process is finally recorded in a wavelet-tree-like structure [33] to support the application of $\pi()$ and $\pi^{-1}()$.

The simplest of these encoding schemes takes advantage of the *runs* in a permutation π, defined as a maximal range of consecutive positions $[i..j]$ that are already sorted in ascending order. This representation uses at most $n(2 + \mathcal{H}(\texttt{LRuns}))(1 + o(1)) + O(\rho \log n)$ bits to encode a permutation of size n covered by ρ, with `LRuns` the sequence of the lengths of the runs, and supports $\pi()$ and $\pi^{-1}()$ in time $O(1 + \log \rho)$. More sophisticated variants of runs considered by Barbay and Navarro allow better compression in applications where those runs arise. A stricter type of run, namely *strict run*, is defined as a maximal range of positions satisfying $\pi(i + k) = \pi(i) + k$, and the *head* of such run is its first

position. For a permutation of size n covered by τ strict runs and $\rho \leq \tau$ runs, this representation supports $\pi()$ and $\pi^{-1}()$ in time $O(1+\log \rho)$ and uses at most $\tau\mathcal{H}(\texttt{LHRuns})(1 + o(1)) + 2\tau \log \frac{n}{\tau} + o(n) + O(\tau + \rho \log \tau)$ bits, where LHRuns is the sequence of the lengths of the runs in the sequence of strict run heads.

Although the performance of these representations is based on the ascending subsequences found inside the permutation, we can use other regularities to develop compressed representations that allow access to the permutation and its inverse without decompressing it.

Barbay *et al.* [4] proposed a new approach for compression of permutations that are partially ordered. They introduced a new measure of presortedness for permutation using the LRM-Tree algorithm [27], and later merged the blocks in the same fashion of Barbay and Navarro [5]. For a permutation of size n partitioned in υ subsequences of lengths LLRM, Barbay *et al.* [4] representation supports $\pi()$ and $\pi^{-1}()$ in time $O(1 + \log \upsilon)$ and uses at most $(1 + \mathcal{H}(\texttt{LLRM}))(n + o(n)) + O(\upsilon \log n)$ bits. Although actually $\rho = \upsilon$, in comparison to Runs-sorting [5], the partition obtained by LRM-sorting [4] improves the merge algorithm since $\mathcal{H}(\texttt{LLRM}) \leq \mathcal{H}(\texttt{LRuns})$, which can yield to a better encoding of the permutation.

4 Thesis Proposal

The main goal of this thesis is to propose compressed data structures inspired by adaptive algorithms in the comparison model, within a scope including permutations, sets, and integers.

4.1 Specific Objectives

The specific objectives aim to design and develop compressed data structures for permutations, sets, and integers, with low space requirements for main memory while efficiently supporting operations.

- Explore the relation between algorithms in the comparison model and useful compressed representations.
- Study the relation between measures of difficulty of an instance and its compressibility.
- Develop new compression schemes based on adaptive algorithms.
- Develop new compressed data structures with efficient support for operations.
- Evaluate the practical applications of the obtained compressed data structures.

4.2 Preliminary Developments

The review of our preliminary developments on compressed data structures for permutations discussed in Section 3.3, is given in Appendix A.

5 Deliverables

The main contributions of this thesis are expected to be:

- New compressed data structures for permutations, sets and integers.
- New measures of compressibility for permutations, sets and integers.
- Implementations of our data structures freely available under a open-source license.

We aim to publish our theoretic and practical results in at least three international conferences.

References

1. B. T. Ávila and E. S. Laber. Merge source coding. In *Proc. 2009 IEEE International Symposium on Information Theory (ISIT)*, pages 214–218, Piscataway, NJ, USA, 2009. IEEE Press.
2. R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
3. J. Barbay, C. Bedregal, and G. Navarro. Improving text indexes using compressed permutations. In *Proc. XIX Encuentro Chileno de Computación, Jornadas Chilenas de Computación*, Antofagasta, Chile, 2010.
4. J. Barbay, J. Fischer, and G. Navarro. LRM-Trees: Compressed indices, adaptive sorting, and compressed permutations. In *Proc. 22th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS, 2011. To appear.
5. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122. Schloss Dagstuhl, Leibnitz Zentrum fuer Informatik, Germany, 2009.
6. C. Bedregal and E. Tellez. Operators over compressed integer encodings. In *Proc. XIX Encuentro Chileno de Computación, Jornadas Chilenas de Computación*, Antofagasta, Chile, 2010.
7. D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, December 2005.
8. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.
9. P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proc. 13th International Conference on World Wide Web (WWW)*, pages 595–602, New York, NY, USA, 2004. ACM.
10. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146. ACM Press, 2008.
11. G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proc. International Conference on Web search and web data mining*, WSDM '08, pages 95–106, New York, NY, USA, 2008. ACM.
12. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
13. T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

14. D. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, Canada, 1996.
15. F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116. Springer, 2007.
16. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer, 2008.
17. O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In C. Álvarez and M. Serna, editors, *Experimental Algorithms*, volume 4007 of *Lecture Notes in Computer Science*, pages 134–145. Springer Berlin / Heidelberg, 2006.
18. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.
19. K. Dudzinski and A. Dydek. On a stable minimum storage merging algorithm. *Inf. Process. Lett.*, 12(1):5–8, 1981.
20. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194 – 203, mar 1975.
21. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.
22. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009. 30 pages.
23. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.
24. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
25. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 150–160. Springer, 2004.
26. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.
27. J. Fischer. Optimal succinctness for range minimum queries. In A. López-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 158–169. Springer Berlin / Heidelberg, 2010.
28. L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2:611–639, October 2006.
29. S. W. Golomb. Run-length encodings. In *IEEE Transactions on Information Theory, IT 12(3)*, pages 399–401, 1966.
30. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 368–373, New York, NY, USA, 2006. ACM.
31. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

32. S. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 210–211. Springer, 2004.
33. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
34. A. Gupta, W.-K. Hon, R. Shah, and S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Data Compression Conference*, 0:213–222, 2006.
35. T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.
36. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.
37. F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.
38. G. Jacobson. Space-efficient static trees and graphs. *Annual IEEE Symposium on Foundations of Computer Science*, 0:549–554, 1989.
39. H. Kamal, S. M. Mirtaheri, and A. Wagner. Scalability of communicators and groups in mpi. In *Proc. 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275, New York, NY, USA, 2010. ACM.
40. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal of Computing*, 1986. 15(1):287–299.
41. S. T. Klein and D. Shapira. Searching in compressed dictionaries. In *Proc. Data Compression Conference*, DCC '02, pages 142–, Washington, DC, USA, 2002. IEEE Computer Society.
42. D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
43. N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Conference on Data Compression*, DCC '99, page 296, Washington, DC, USA, 1999. IEEE Computer Society.
44. A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
45. V. Mäkinen. Compact suffix array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1848, pages 305–319, 2000.
46. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
47. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. Special issue on "The Burrows-Wheeler Transform and its Applications".
48. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22:935–948, October 1993.
49. H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34(4):318–325, 1985.
50. G. Manzini. An analysis of the burrows—wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
51. K. Mehlhorn. Sorting presorted files. In Springer, editor, *Proc. 4th GI-Conference on Theoretical Computer Science*, volume 67 of *Lecture Notes in Computer Science*, pages 199–212, 1979.

52. A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.
53. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
54. J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.
55. J. I. Munro, R. Raman, V. Raman, and S. Rao. Succint representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 345–356. Springer-Verlag, 2003.
56. J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, page 118, Washington, DC, USA, 1997. IEEE Computer Society.
57. G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
58. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
59. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
60. R. F. Rice. Some practical universal noiseless coding techniques. In *Jet Propulsion Laboratory, Pasadena, California, JPL Publication 79–22*, 1976.
61. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 225–232, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.
62. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
63. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.
64. I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, Inc., New York, NY, USA, 1994.
65. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

APPENDIX

A Preliminary Developments on Compression of Permutations

The compressed data structures proposed by Barbay and Navarro [5] were applied in problems of scalability in MPI (Message Passing Interface) applications [39] and text indexing for large text collections [3] with promising results. For the latter, we evaluated the application of compressed representations for permutations in two scenarios of text indexing: inverted indexes and suffix arrays, as described in the section that follows.

A.1 Inverted Indexes

Inverted indexes are very popular for text retrieval in natural language [2]. We consider a text $T[1, n]$ of n words with ρ distinct words. The concatenation of the ρ inverted lists represents a permutation of $[1..n]$ with at most ρ runs, so it is possible to compress it using the schemes for compression of permutations by Barbay and Navarro [5].

We evaluated the compression techniques based on runs (`RunIndex`) and strict runs (`SRunIndex`) in comparison to `WPH` [10], a competitive text index that improves over the Plain Huffman coder [53]. 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 [22]. Table 1 shows some statistics of the text.

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

Table 1. Description of the text used for natural language.

Table 2 shows the compression ratio obtained by each technique. The amount of memory usage of the `RunIndex` and `SRunIndex` encodings are similar to that required by `WPH`. Although `RunIndex` achieves a better compression, `SRunIndex` does not achieve a good ratio because of the lack of strict runs in the permutation. Statistical measures on the text showed that the average run size is 511 while the average strict run size is 1, which shows how the presence – or absence – of runs in the text directly affects the compression obtained.

Next we examined the performance of the three indexes when searching for words. We compare the time to *locate* all the text occurrences of a pattern and the time to extract a *snippet* of 10 words around each of these occurrences. In both scenarios we consider words from 4 different ranges of frequency. Table

Text	RunIndex	SRunIndex	WPH
english	0.32	0.38	0.36

Table 2. Memory usage of each index (fraction of the original text).

Query	Freq.	RunIndex	SRunIndex	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 3. Performance of the indexes for different word frequencies (times in seconds).

3 shows the average time per pattern from 100 randomly chosen single-word patterns.

Both operations of location and extraction achieved the better times using our compression schemes. For the case of locate, the times of the `WPH` index were close for the most frequent words. For extracting snippets, the `RunIndex` and `SRunIndex` indexes were on average 4 times faster than `WPH`, which is a great advantage considering that the `RunIndex` index requires less space to operate. `RunIndex` and `SRunIndex` indexes obtained the snippets accessing the inverse permutation π^{-1}, while locate queries were performed accessing π. Both indexes are considered self-index since they are capable of reproducing the original text.

A.2 Suffix Arrays

When a text cannot be handled with inverted indexes, suffix arrays are a common strategy for indexing the text. Again, consider a text $T[1, n]$ of n symbols and an 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 $T[A[i+1], n]$. There exist various compressed representations of suffix arrays because of the high space requirement of an uncompressed index [23, 24, 26, 33, 62]. One of our interest, the *Compressed Suffix Array (CSA)* of Sadakane [62], builds over a permutation Ψ of $[1..n]$, where $\Psi(i)$ stores the position in A of the next symbol of suffix $A[i]$. Permutation Ψ lets us navigate one position forward in the text, and along a few extra structures, it can be turned into a self-index. In a similar way, the family of FM-Index [24, 26] also aim to compress the suffix array, but their approach allows a backward navigation of the suffixes instead through the Burrows-Wheeler Transform of the text.

We evaluated the indexes `RunIndex` and `SRunIndex` in this scenario, and compared our results with existing techniques for compression of suffix arrays,

specifically: Compressed Suffix Array (`CSA`) [62], Succinct Suffix Array (`SSA`) [26], Practical Succinct Suffix Array (`FSSA`) [16], Run-Length FM-Index (`RLFMI`) [46] and the Alphabet-Friendly FM-Index (`AFFMI`) [26]. The experiments were performed using four text collections: **`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 [22].

Tables 4 and 5 summarize the statistics about the runs and strict runs respectively, found in the permutation Ψ of each text. The second column of Table 4 shows the total number of runs 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. In Table 5, the second column shows the total number of strict runs in Ψ, the third column shows the entropy of the distribution of the run lengths in the permutation of strict run heads (`HLRuns`), 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}).

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 4. 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 5. Statistics of strict runs in permutation Ψ of the texts.

Tables 4 and 5 can help to understand the effect of the distribution of runs and strict runs in the performance of our indexes. For example, the entropy values of `LRuns` and `LHRuns` indicate that the sorting algorithm performs better than using a traditional balanced-merge strategy, as both are inferior than $\log \rho$.

Table 6 summarizes the memory usage of the indexes considering only the data structures needed to support $\Psi()$ for `CSA`, or $\Psi^{-1}()$ for `AFFMI`, `RLFMI`, `SSA`, `FSSA`. For `RunIndex` and `SRunIndex`, the usage shown is enough to directly support both operations.

Text	RunIndex	SRunIndex	CSA	AFFMI	RLFMI	SSA	FSSA
dna	0.42	0.52	0.46	0.48	0.75	0.42	0.43
proteins	0.58	0.57	0.67	0.82	0.89	0.69	0.69
sources	0.74	0.44	0.38	0.73	0.74	0.85	0.54
xml	0.71	0.37	0.29	0.54	0.65	0.83	0.47

Table 6. Memory usage of the indexes expressed as a fraction of the original text.

For the `sources` and `xml` texts, `SRunIndex` index achieved better compression because the strict runs 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 `RunIndex` 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 $\mathcal{H}(\texttt{LRuns})$.

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 (`RunIndex`) and strict runs (`SRunIndex`) to the `CSA` index, which compresses the suffix array via the function Ψ that captures text regularities and allows forward navigation inside the text. As shown in Figure 1, the performance of `CSA` is better than `RunIndex` and `SRunIndex` indexes in every scenario, since `CSA` also takes advantage of the ascending runs present in Ψ.

Even when `CSA` performs better in time, `RunIndex` and `SRunIndex` indexes do not depend as much on sampling parameters as `CSA` does (S_Ψ could be modified to reduce the space, but it would negatively affect the access time to Ψ). In contrast to `CSA`, both `RunIndex` and `SRunIndex` behave as a bidirectional index since they take advantage of present runs to improve 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 (`RunIndex`) and strict runs (`SRunIndex`) to the group of indexes from the FM-Index family: Succinct Suffix Array (`SSA`), Practical Succinct Suffix Array (`FSSA`), Run-Length FM-Index (`RLFMI`) and the Alphabet-Friendly FM-Index (`AFFMI`). These indexes are built using the Burrows-Wheeler Transform (BWT) [12] and backward searching, allowing backward navigation inside the text.

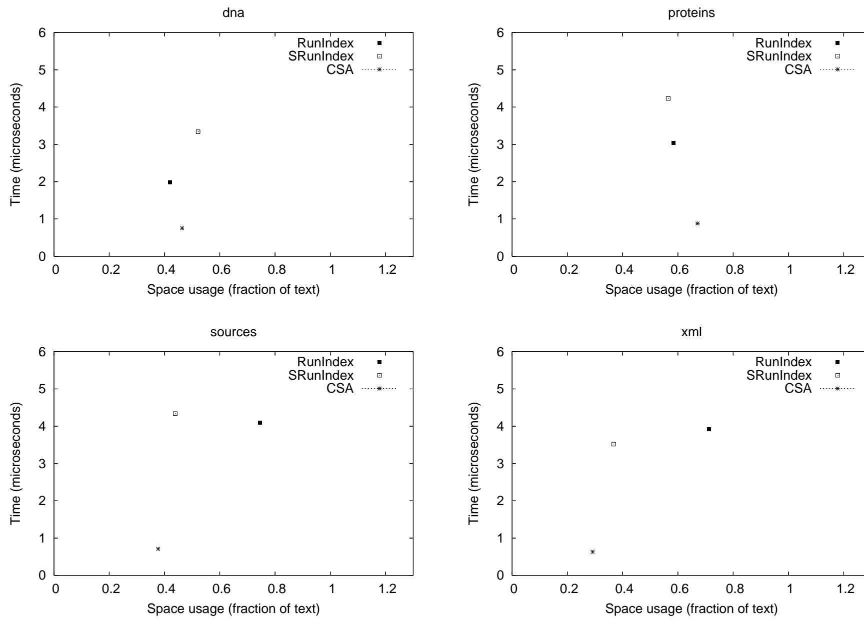

Fig. 1. Space-time tradeoffs for evaluating Ψ.

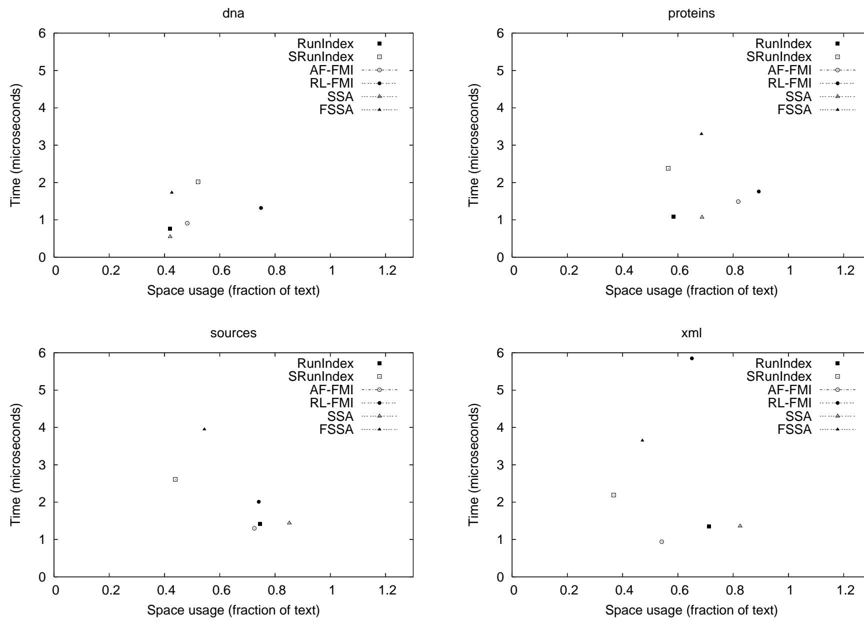

Fig. 2. Space-time tradeoffs for evaluating Ψ^{-1} (LF).

RunIndex and SRunIndex indexes achieved competitive times calculating Ψ^{-1} within a comparable space requirement. The aforementioned observations about the runs distribution still apply to this scenario as RunIndex and SRunIndex indexes remain the same as in the previous experiment: the distribution of runs inside the text also helps to improve backward navigation.

Figures 1 and 2 illustrated the ability of RunIndex and SRunIndex indexes to allow both forward and backward navigation inside the text in competitive times, a feature useful in operations that require, for example, random access to the text or extraction of snippets of variable lengths.

Implications for further study From the preliminary results discussed in this section, we plan to work in the following tasks:

- Perform more experiments on the performance of the obtained text indexes.
- Evaluate the performance of the runs-based compressed data structures [5] in other scenarios such as the representation of sequences of Golynsky *et al.* [30].
- A practical evaluation of the LRM data structure [4] in text indexes.
- Design of improvements in the merge step of the sorting algorithms evaluated.
- Evaluate how to use 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), to develop new compressed data structures for permutations.