Locally Compressed Suffix Arrays

Rodrigo González, Gonzalo Navarro, and Héctor Ferrada
Dept. of Computer Science, University of Chile.
{rgonzale,gnavarro,hferrada}@dcc.uchile.cl

We introduce a compression technique for suffix arrays. It is sensitive to the compressibility of the text and *local*, meaning that random portions of the suffix array can be decompressed by accessing mostly contiguous memory areas. This makes decompression very fast, especially when various contiguous cells must be accessed.

Our main technical contributions are the following. First, we show that runs of consecutive values that are known to appear in function $\Psi(i) = A^{-1}[A[i]+1]$ of suffix arrays A of compressible texts also show up as repetitions in the differential suffix array $A'[i] = A[i] - A[i-1]$. Second, we use Re-Pair, a grammar-based compressor, to compress the differential suffix array, and upper bounds its compression ratio in terms of the number of runs. Third, we show how to compact the space used by the grammar rules by up to 50%, while still permitting direct access to the rules. Fourth, we develop specific variants of Re-Pair that work using knowledge of Ψ, and use much less space than the general Re-Pair compressor while achieving almost the same compression ratios. Fifth, we implement the scheme and compare it exhaustively with previous work, including the first implementations of previous theoretical proposals.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*; H.2.1 [**Database management**]: Physical design—*Access methods*; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Compressed suffix arrays, Re-Pair, Text databases

1. INTRODUCTION

Suffix trees [Weiner 1973; McCreight 1976] and suffix arrays [Gonnet et al. 1992; Manber and Myers 1993] are probably the most important data structures in stringology, with myriad widely recognized virtues and applications [Apostolico 1985; Gusfield 1997; Crochemore and Rytter 2002]. Their most serious drawback is their space usage. Over a text $T[1,n] = t_1 t_2 \dots t_n$ on alphabet $[1,\sigma]$, which can be represented within $n\lceil \lg \sigma \rceil$ bits, these structures require $\Theta(n \lg n)$ bits. In practice, suffix trees are 10 to 20 times larger than the text [Kurtz 1998], whereas suffix arrays are at least 4 times larger.

A suffix array is essentially a permutation $A[1,n]$ listing all the suffixes $T[i,n] =$

Partially supported by Fondecyt grant 1-110066, Chile. An early partial version of this article appeared in *Proc. 18th Combinatorial Pattern Matching (CPM)*, LNCS 4580, pp. 216–227, 2007.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

$t_i t_{i+1} \dots t_n$ in lexicographic order. A suffix tree is a trie (or digital tree) storing all those suffixes, and its leaves correspond to the suffix array. Suffix trees are more powerful than suffix arrays, but it has been shown that suffix trees can be represented using little extra space on top of the corresponding suffix array [Kärkkäinen 1995; Abouelhoda et al. 2004; Sadakane 2007; Fischer et al. 2009; Fischer 2010; Russo et al. 2011]. Representing the suffix array is thus the main memory bottleneck, and the focus of this paper. We seek to *compress* the suffix array in a way that it can still be accessed at random positions in compressed form (a plain compression per se is meaningless, as the suffix array can in principle be derived from the compressed text).

Two of the most basic functionalities offered by suffix arrays are *counting* and *locating*. Counting refers to computing the number of times a pattern $P[1,m]$ appears in T. This is done by identifying the area $A[sp,ep]$ where the suffixes start with P; then P appears $ep - sp + 1$ times in T. This can be done in time $O(m \lg n)$ over a plain representation of A, via two binary searches (this decreases to $O(m + \lg n)$ if using more space than the bare permutation, and $O(m)$ if using a suffix tree). Locating refers to giving the positions where P appears in T. This is done by listing the values $A[i]$, $sp \leq i \leq ep$, and can be done in $O(1)$ time per occurrence on a plain representation of A.

Current compressed suffix arrays achieve a counting performance that is comparable, in theory and in practice, with that of plain suffix arrays [Navarro and Mäkinen 2007; Ferragina et al. 2009; Belazzougui and Navarro 2011]. Locating, on the other hand, is far behind, hundreds to thousands of times slower than the plain structure. While the plain suffix array requires $O(1)$ time (just a memory access) to compute the value of a cell $A[i]$, most compressed representations require $O(\lg^\epsilon n)$ time, where ϵ is in practice larger than 1. Worse than that, the memory access patterns of the compressed solutions are highly non-local.

In this paper we propose a new suffix array compression technique we dub *locally compressed suffix array (LCSA)*, based on grammar compression. The LCSA requires $O(\rho(1 + \lg \frac{n}{\rho}) \lg n + n \lg^{1-\epsilon} n)$ bits, where $\rho \leq n$ is the number of *runs* in A. This measure [Mäkinen and Navarro 2005; Navarro and Mäkinen 2007] is smaller as T is more compressible. The LCSA can extract any c consecutive cells $A[i, i+c-1]$ in time $O(c + \lg^\epsilon n \lg\lg n)$. Being faster to extract consecutive cells is relevant, in particular, when displaying the $c = ep - sp + 1$ occurrences of a pattern P in $A[sp,ep]$.

In practice, the LCSA is shown to reduce the suffix array to as little as 25% of its original size, depending on the compressibility of the text type. Using the same space, the LCSA is much faster than alternative compressed suffix arrays for accessing more than a few consecutive cells ($c > 2$–6). Compared to a plain suffix array, accessing the LCSA is about 10 times slower. Its local decompression makes the LCSA a good candidate to be deployed on disk as well.

In technical terms, our contributions are the following:

(1) We design a suffix array compression scheme that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible (more precisely, the so-called runs [Mäkinen and Navarro 2005] of the suffix array). These regularities have been exploited in several ways in the

past [Mäkinen 2003; Grossi and Vitter 2005; Grossi et al. 2003; Sadakane 2003; Mäkinen and Navarro 2005; Mäkinen et al. 2010], but we present a completely novel technique to take advantage of it. We represent the suffix array using differential encoding, which converts the regularities into true repetitions. Those repetitions are then factored out using Re-Pair [Larsson and Moffat 2000], a grammar-based compression technique that offers fast local decompression.

(2) By studying the connection between the repetitions in the differential suffix array and the runs in A, we prove that the application of Re-Pair to the differential suffix array leads to $O(\rho \lg \frac{n}{\rho})$ elements in the compressed sequence, which is the key to the space analysis of the LCSA.

(3) We introduce a representation of the Re-Pair rules that reduces their space to up to 50%, while retaining direct access to rules.

(4) We use specific suffix array properties to use less space than Re-Pair to compress. Our technique deviates from Re-Pair by using Ψ to guide the formation of rules. We prove that this method obtains the same asymptotic performance of Re-Pair and that, in practice, it loses at most 5% of compression ratio.

(5) We implement the LCSA and exhaustively study its practical behavior. We also compare it with various alternative compressed suffix arrays, some of which are already implemented, while for others we provide their first implementation.

Obtaining compressed indexes able to count in competitive time was the first important breakthrough in this area. We believe our work is a first important step towards compressed indexes with practical locating times. This is up to date the major concern for adopting compressed indexes in practical applications.

This paper is organized as follows. In Section 2 we review the related work on compressed suffix arrays and put our contribution in context. In Section 3 we describe our LCSA. In Section 4 we carry out several experimental tunings and comparisons with alternative plain and compressed suffix arrays. In Section 5 we give our conclusions and future work directions. A theoretical analysis of the compression achieved by the LCSA is given in Appendix A.

2. RELATED WORK

One of the earliest attempts to compress suffix arrays was to sample some of their entries. A regular text sampling [Kärkkäinen and Ukkonen 1996b] only stores the suffixes that start at a position multiple of q. This reduces the suffix array space to $O((n/q) \lg n)$ and still can count and locate the occurrences of any pattern of length q or more. By choosing a more sophisticated text sampling [Kärkkäinen and Ukkonen 1996a], patterns of any length can be handled, and the suffix array space is $O(n \lg \sigma)$ bits, that is, of the same order required for the plain storage of T. Other samplings have been proposed, see for example Claude et al. [2012]. While adequate for the basic counting and locating tasks, these sampling approaches are unable to retrieve a random cell $A[i]$ of the suffix array, and thus cannot be used in more complex scenarios where suffix arrays are useful (for example, regular expression search [Baeza-Yates and Gonnet 1996] or approximate search [Ukkonen 1993] on suffix trees, which can be run on suffix arrays).

The first *complete* representation of suffix arrays (i.e., able to return any cell $A[i]$) achieving reduced space was proposed by Grossi and Vitter [2000; 2005]. They achieve $(\frac{1}{\epsilon} + O(1))n \lg \sigma$ bits of space and can access any suffix array cell in time $O(\lg_\sigma^\epsilon n)$. In addition, they support counting in time $O(\frac{m}{\lg_\sigma n} + \lg_\sigma^\epsilon n)$. Shortly after, Rao [2002] used a similar scheme to obtain a range of space/time tradeoffs, using $O((t \lg_\sigma^{1/t} n) n \lg \sigma)$ bits and accessing any cell in time $O(t)$, for any $1 \leq t \leq \lg \lg_\sigma n$ (see Appendix D for this analysis on a general alphabet).

Mäkinen [2000; 2003] pioneered the *opportunistic* compression of suffix arrays, that is, compression schemes that use less space on compressible texts. Here it is useful to define function $\Psi(i) = A^{-1}[(A[i] \bmod n)+1]$, which is another permutation that we will use later. A *run* is a maximal contiguous subsequence of A with consecutive Ψ values. On a random text, A is covered by $\rho = \Theta(n)$ runs, but on a compressible text, ρ can be much lower [Mäkinen and Navarro 2005; Mäkinen et al. 2010]. It was shown [Navarro and Mäkinen 2007] that the technique of Mäkinen [2000; 2003] uses $2\rho \lg n + O(n \lg^{1-\epsilon} n)$ bits and can access any cell in time $O(\lg^{2\epsilon} n)$ (by setting its parameters to $C = D = \lg^\epsilon n$). It can count in time $O(m \lg n)$. Value ρ can be upper bounded with $\rho \leq \min(n, nH_k + \sigma^k)$ for any k [Mäkinen and Navarro 2005], but it can be much smaller. Here H_k is the k-th order empirical entropy of T, a measure of its statistical compressibility [Manzini 2001]: for any $k > 0$ it holds $H_k \leq H_0 \leq \lg \sigma$.

Almost simultaneously, Ferragina and Manzini [2000; 2005] proposed the first compressed *self-index*, which is a compressed suffix array that can also reproduce any substring of T. On constant-size alphabets, the structure uses $O(nH_k) + O(n/\lg^\epsilon n)$ bits for any $k \leq \alpha \lg_\sigma n$ and constant $0 < \alpha < 1$. The structure can retrieve any entry of A in time $O(\lg^\epsilon n)$. It uses a novel technique for determining the range $[sp, ep]$ that allows it to count in time $O(m)$. A later generalization for general alphabets [Ferragina et al. 2007; Belazzougui and Navarro 2011] reached $nH_k + O(n \lg \sigma / \lg^\epsilon n)$ bits of space, $O(m)$ counting time, and $O(\lg^{1+\epsilon} n)$ time to access any cell.

The structure of Grossi and Vitter [2000] also evolved into a compressed self-index. Sadakane [2000; 2003] obtained $\frac{1}{\epsilon} nH_0 + O(n \lg H_0)$ bits and $O(\lg^\epsilon n)$ time to access a cell, and $O(m \lg n)$ counting time. Grossi et al. [2003] improved the representation and achieved various tradeoffs. One requires $(1 + \frac{1}{\epsilon})nH_k + O(n)$ bits and $O(\lg_\sigma^\epsilon n + \lg \sigma)$ time to access a cell. A faster one requires $nH_k \lg \lg_\sigma n + O(n)$ bits and $O(\lg \lg_\sigma n + \lg \sigma)$ access time, and a smaller one requires $nH_k + O(n \lg \sigma / \lg^\epsilon n)$ bits and $O(\lg^{1+\epsilon} n)$ access time. Counting times are $O(m/\lg_\sigma n + \mathrm{polylog}(n))$.

Our LCSA uses $O(\rho \lg \frac{n}{\rho} \lg n)$ bits, plus a sampling using $O(n \lg^{1-\epsilon} n)$ bits, for any $\epsilon > 0$. It retrieves any cell in time $O(\lg^\epsilon n \lg \lg n)$, and c consecutive cells in time $O(c + \lg^\epsilon n \lg \lg n)$. Table I summarizes all the complexities and puts our contribution in (theoretical) context.

3. THE LOCALLY COMPRESSED SUFFIX ARRAY (LCSA)

In this section we describe our LCSA data structure. The theoretical analysis of its space usage is given in Appendix A.

3.1 Technical Preliminaries

Given a text $T = t_1 \ldots t_n$ over alphabet Σ of size σ, where for technical reasons we assume $t_n = \$$ is smaller than any other character in Σ and appears nowhere else

Source	Space in bits	Access time
[Manber and Myers 1993]	$n \lg n$	$O(1)$
[Grossi and Vitter 2000; 2005]	$(\frac{1}{\epsilon} + O(1)) n \lg \sigma$	$O(c \cdot \frac{1}{\epsilon} \lg_\sigma^\epsilon n)$
[Rao 2002]	$O(t \lg_\sigma^{1/t} n) n \lg \sigma$	$O(c \cdot t)$
[Mäkinen 2000; 2003]	$2\rho \lg n + O(n \lg^{1-\epsilon} n)$	$O(c \cdot \lg^{2\epsilon} n)$
[Ferragina and Manzini 2000; 2005]	$O(nH_k) + O(n / \lg^\epsilon n)$	$O(c \cdot \lg^\epsilon n)$
[Ferragina et al. 2007]	$nH_k + O(n \lg \sigma / \lg^\epsilon n)$	$O(c \cdot \lg^{1+\epsilon} n)$
[Sadakane 2003]	$\frac{1}{\epsilon} nH_0 + O(n \lg H_0)$	$O(c \cdot \lg^\epsilon n)$
[Grossi et al. 2003]	$(1 + \frac{1}{\epsilon}) nH_k + O(n)$	$O(c(\lg_\sigma^\epsilon n + \lg \sigma))$
	$nH_k \lg \lg_\sigma n + O(n)$	$O(c(\lg \lg_\sigma n + \lg \sigma))$
	$nH_k + O(n \lg \sigma / \lg^\epsilon n)$	$O(c \cdot \lg^{1+\epsilon} n)$
LCSA	$O(\rho \lg \frac{n}{\rho} \lg n + n \lg^{1-\epsilon} n)$	$O(c + \lg^\epsilon n \lg \lg n)$

Table I. Plain and compressed suffix arrays, with their space in bits and time to access c consecutive cells. Here $\rho \leq n$ is the number of runs in A, and $\epsilon > 0$ and $1 \leq t \leq \lg \lg n$ are parameters. The complexities for Ferragina and Manzini [2000; 2005] hold only for fixed alphabets, $\sigma = O(1)$.

in T, a *suffix array* $A[1, n]$ is a permutation of $[1, n]$ such that $T_{A[i],n} \prec T_{A[i+1],n}$ for all $1 \leq i < n$, being "$\prec$" the lexicographical order. By $T_{j,n}$ we denote the *suffix* of T that starts at position j. Since all the occurrences of a pattern $P = p_1 \dots p_m$ in T are prefixes of some suffix, a couple of binary searches in A suffice to identify the segment in A of all the suffixes that start with P, that is, the segment pointing to all the occurrences of P. Thus the suffix array permits counting the occurrences of P in $O(m \lg n)$ time and reporting the c occurrences in $O(c)$ time. With an additional array of integers, the counting time can be reduced to $O(m + \lg n)$ [Manber and Myers 1993].

Suffix arrays turn out to be compressible whenever T is. The compressibility of T shows up in A, in particular, in the form of long segments $A[i, i+\ell]$ that appear elsewhere in $A[j, j+\ell]$ with all the values shifted by one position. One can partition A into ρ *runs* of maximal segments that appear repeated (shifted by 1) elsewhere.

Definition 1. *A* run *in the suffix array A of T is a maximal segment $A[i, i+\ell]$ such that there exists another segment $A[j, j+\ell]$, $j \neq i$, such that $A[j+s] = A[i+s] + 1$ for all $0 \leq s \leq \ell$. We define $\rho \leq n$ as the minimum number of runs into which A can be partitioned.*

Our definition of run is closely related to the definition of "runs in Ψ", for function $\Psi(i) = A^{-1}[(A[i] \bmod n) + 1]$ [Grossi and Vitter 2005], which is widely used in compressed text indices [Navarro and Mäkinen 2007]. A *run in* Ψ is a maximal area $\Psi(i, i+\ell)$ where Ψ contains consecutive values.

Definition 2. *A* run *in function Ψ of A is a maximal segment $\Psi(i, i+\ell)$ such that $\Psi(i+s) = \Psi(i+s-1) + 1$ for all $1 \leq s \leq \ell$.*

Lemma 1. *Runs in A are runs in Ψ and vice versa.*

Proof. Let $A[i, i+\ell]$ be a run in A. Then there exists another segment $A[j, j+\ell]$, $j \neq i$, such that $A[j+s] = A[i+s] + 1$ for all $0 \leq s \leq \ell$. Thus $\Psi(i+s) = j+s$ for all $0 \leq s \leq \ell$. Then $\Psi(i+s) - \Psi(i+s-1) = (j+s) + (j+s-1) = 1$ for all $1 \leq s \leq \ell$ and thus $\Psi(i, i+\ell)$ is a run in Ψ. Conversely, if $\Psi(i+s) = \Psi(i+s-1)+1$ for all $1 \leq s \leq \ell$, then $A[j+s] = A[i+s]+1$ for $j = \Psi(i)$ for all $0 \leq s \leq \ell$ and thus

$A[i, i+\ell]$ is a run in A. Note that the value i such that $\Psi(i) = 1$ can only appear at the beginning of a run in Ψ. $\square$

The k-th order empirical entropy of T, H_k [Manzini 2001], is the minimum number of bits per symbol emitted by a statistical compressor that encodes each symbol of T with a code that depends only on the symbol and the k symbols that precede it. It is used to measure the performance of various text compressors and compressed text indexes. Interestingly, it has been shown that this measure is related to the number of runs in Ψ by the upper bound $\rho \leq nH_k + \sigma^k$ for any k [Mäkinen and Navarro 2005; Navarro and Mäkinen 2007]. Yet, especially on repetitive texts, ρ can be much smaller [Mäkinen et al. 2010].

The runs in Ψ have been used several times in the past to compress A. Mäkinen's Compact Suffix Array (CSA) [Mäkinen 2003] replaces runs with pointers to their definition (copy) elsewhere in A, so that the run can be recovered by (recursively) expanding its definition and shifting the values. Mäkinen and Navarro [2005] use the connection with FM-indexes (runs in A are related to equal-letter runs in the Burrows-Wheeler transform of T, a basic building block of FM-indexes) and run-length compression. Various other authors have run-length compressed Ψ directly [Grossi et al. 2003; Sadakane 2003; Mäkinen et al. 2010].

3.2 Basic LCSA Idea

We present a completely different method to compress A based on its runs. We first represent A in differential form A':

Definition 3. *Let $A[1, n]$ be an array of integers. Then we define $A'[1, n]$ as follows: $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$ for all $1 < i \leq n$.*

The next simple lemma shows that runs of A become true repetitions in A'.

Lemma 2. *Consider a run of A of the form $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Then $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$.*

Proof. $A'[j+s] = A[j+s] - A[j+s-1] = (A[i+s]+1) - (A[i+s-1]+1) = A[i+s] - A[i+s-1] = A'[i+s]$. $\square$

We can now exploit those repetitions using any classical compression method. In particular, we seek a method that allows fast local decompression of A'.

We resort to Re-Pair [Larsson and Moffat 2000], a grammar-based compression method based on the following algorithm:

(1) Identify the most frequent pair $A'[i]A'[i+1]$ in A', let ab be such pair.
(2) Create a new integer symbol s, larger than all existing symbols in A', and add rule $s \rightarrow ab$ to a dictionary R.
(3) Replace every occurrence of ab in A by s.[1]
(4) Iterate until every pair has frequency 1.

The result of the compression algorithm is the dictionary of rules R plus the sequence C of (original and new) symbols into which A' has been compressed.

[1]If $a = b$ it might be impossible to replace all occurrences, e.g. aa in aaa. But, in such case, one can at least replace each other occurrence in a row.

Note that R can be easily stored as a vector of pairs: the numbers s assigned start at $n+1$, which is larger than any possible value in A' (note $A'[1] = A[1] = n$ is the largest value in A'), and rule $s \rightarrow ab$ can be stored at $R[s-n] = a : b$.

Any portion of C can be easily decompressed in optimal time and fast in practice. To decompress $C[i]$, we first check if $C[i] \leq n$. If it is, then it is an original symbol of A' and we are done. Otherwise, we obtain both symbols from $R[C[i]-n]$, and expand them recursively (they can in turn be original or created symbols, and so on). We reproduce u cells of A' in $O(u)$ time.

As R grows by 2 integers $a : b$ for every new pair, we may stop creating pairs when the most frequent one appears only twice. R can be further reduced by preempting the compression process, which trades the size of R for overall compression ratio.

We need a few more structures on top of R and C in order to provide direct access to the values of A:

—An array S such that $S[i] = A[i \cdot l]$, that is, a sampling of absolute values of A at regular intervals l.

—A bitmap $L[1,n]$, marking the positions where each symbol of C (which could represent several symbols of A') starts in A'.

—$o(n)$ further bits to answer *rank* queries on L in constant time [Munro 1996; Clark 1996], where $rank(L,i)$ is the number of 1's in $L[1,i]$.

With this structure, the algorithm to retrieve $A[i, i+c-1] = A[i,j]$ is as follows:

(1) If necessary, extend the interval $[i,j]$ to $[i',j']$ the minimum necessary so that a multiple of l is included in $[i',j']$. Note that only i or j must be extended, by at most $\lfloor l/2 \rfloor$ positions.

(2) Use the mechanism to decompress one symbol in C (described above) to obtain $A'[i',j']$, by expanding $C[rank(L,i'), rank(L,j')]$. In practice, if we expand rules left to right, the last symbol of C needs not be fully expanded.

(3) Use any absolute sample of A included in $S[\lceil i'/l \rceil, \lfloor j'/l \rfloor]$ to obtain, using the differences in $A'[i',j']$, the values $A[i',j']$.

(4) Return the values in the original requested interval $A[i,j]$.

The overall time complexity of this decompression is the output size plus the amount we have expanded the interval to include a multiple of l and to expand an integral number of symbols in C. The former is at most $l/2$. The latter can be controlled by limiting the length of any nonterminal to a maximum of d original symbols. Then the cost to access c contiguous cells is $O(l+d+c)$.

Operationally, to limit the length of nonterminals, it is sufficient to insert distinct special symbols at positions multiple of d in T, since Re-Pair will not pair them.

3.3 Compression using Ψ: Algorithm RPΨ_0

A weak point of using Re-Pair is its space usage. While it can be implemented in $O(n)$ time, it requires to store, at the very least, 3 integers per original text symbol (and at most many more, as it stores several integers per distinct pair in A') [Larsson and Moffat 2000]. In this section we introduce a fast approximate technique specialized in compressing differential suffix arrays A'. We show that Ψ (which is easily built in $O(n)$ time from A) can be used to obtain a fast compression algorithm that requires two integers per original symbol and in practice compresses almost as well as the original Re-Pair.

Recall that $\Psi(i)$ tells where in A is the value $A[i]+1$. The idea is that, if $A[i,i+\ell]$ is a run such that $A[j+s]=A[i+s]+1$ for $0 \le s \le \ell$ (and thus $A'[j+s]=A'[i+s]$ for $1 \le s \le \ell$), then $\Psi(i+s)=j+s$ for $0 \le s \le \ell$. Thus, by following permutation Ψ we have a good chance of finding repeated pairs in A'. The basic idea is to choose the pairs while following permutation Ψ, cycling several times over A', until no further replacements can be done. This does not guarantee to choose the same pairs of the original Re-Pair, but we expect them to be sufficiently good (we will later confirm this expectation, both analytically and experimentally).

Data Structures. To compress using Ψ we need two arrays and one bitmap. One of the arrays is the very same input array A', which is reused and converted into the output array C. The other array is Ψ, which is also overwritten. The bitmap is the output bitmap L. More precisely, we use:

—An array $D[1,n]$, which initially stores the suffix array of text T in differential form, $D[i]=A'[i]$ for all i. At the end, we compact the valid values of D to obtain C.
—An array $P[1,n]$, which initially stores the values of function Ψ of text T, $P[i]=\Psi(i)$, for all i.
—The bitmap $L[1,n]$, where $L[i]=1$ indicates that $D[i]$ is a valid value. In the beginning $L[i]=1$ for all i. At the end, L can be preprocessed for *rank* queries and is ready for querying.

Note that the dictionary R is produced via appends and never read, thus it does not need to be maintained in main memory but just output to disk as we compress.

When we replace a pair with a new symbol, array D becomes sparse. A way to find the next valid symbol in constant time is to maintain the following invariant: If a valid symbol $D[i]$ is followed by an invalid symbol $D[i+1]$ (that is, $L[i]=1$ and $L[i+1]=0$), then $D[i+1]$ stores the position i' of the next valid symbol $D[i']$ (we use i' with this meaning, for any i, in the next algorithm description).

In practice, it turns out that, faster than maintaining $D[i+1]=i'$, is to calculate $i'=\mathit{selectnext}(L,i)$, which returns the position of the first 1 in $L[i+1,n]$ by scanning the bitmap word-wise.

Algorithm. We make a number of *passes* over D. Each pass starts at $i=1$ (where value $A'[1]=A[1]=n$ will not be replaced by Re-Pair, as it is unique). For each i visited along the pass, we set $a:b=D[i]:D[i']$ and see if $a:b=D[P[i]]:D[P[i]']$. If this does not hold, we move on to $i \leftarrow P[i]$ and read a new pair $a:b$. If, instead, equality holds, we start a chain of replacements: We add a new pair $s \rightarrow a:b$ to R, make the replacements at i and $P[i]$ (invalidating i' and $P[i]'$), and move on to $i \leftarrow P[i]$, continuing the replacements until we reach a pair $D[P[i]]:D[P[i]'] \neq a:b$. Then we restart the process with $i \leftarrow P[i]$, looking again for a new pair $a:b$.

To invalidate position i' we set $L[i'] \leftarrow 0$. Then, if $L[i'+1]=1$, we set $D[i+1] \leftarrow i'+1$, else we set $D[i+1] \leftarrow D[i'+1]$. This maintains the invariant (if we use *selectnext*, setting $L[i']$ to zero is sufficient). Moreover, let $i'=P[k]$, then we set $P[k] \leftarrow P[P[k]]$, so that position i' is skipped in the next pass. We proceed analogously to invalidate $P[i]'$.

```
Algorithm RPΨ_0(D, P, α)
    $s \leftarrow n$, $R \leftarrow \emptyset$
    for $i \leftarrow 1 \ldots n$ do $L[i] \leftarrow 1$
    $n' \leftarrow n$
    do $rep \leftarrow 0$
        $j \leftarrow 1$, $j' \leftarrow next(j)$
        do
            do $i \leftarrow j$, $i' \leftarrow j'$
                while $L[P[i]] = 0$ do $P[i] \leftarrow P[P[i]]$
                $j \leftarrow P[i]$, $j' \leftarrow next(j)$
            while $j \neq 1$ and $D[i] : D[i'] \neq D[j] : D[j']$
            if $j \neq 1$ then
                $a : b \leftarrow D[i] : D[i']$, $s \leftarrow s + 1$, $R \leftarrow R \cup \{s \rightarrow ab\}$
                $D[i] \leftarrow s$, $L[i'] \leftarrow 0$, $rep \leftarrow rep + 1$
                if $L[i'+1] = 1$ then $D[i+1] \leftarrow i'+1$ else $D[i+1] \leftarrow D[i'+1]$ //invalidate
                do $D[j] \leftarrow s$, $L[j'] \leftarrow 0$, $rep \leftarrow rep + 1$
                    if $L[j'+1] = 1$ then $D[j+1] \leftarrow j'+1$ else $D[j+1] \leftarrow D[j'+1]$ //invalidate
                    $i \leftarrow j$, $i' \leftarrow j'$
                    while $L[P[i]] = 0$ do $P[i] \leftarrow P[P[i]]$
                    $j \leftarrow P[i]$, $j' \leftarrow next(j)$
                while $j \neq 1$ and $a : b = D[j] : D[j']$
        while $j \neq 1$
        $n' \leftarrow n' - rep$
    while $(rep > \alpha(n' + rep))$
    $j \leftarrow 1$
    for $i \leftarrow 1 \ldots n$ do
        if $L[i] = 1$ then $D[j] \leftarrow D[i]$, $j \leftarrow j + 1$
    return $(C[1, n'] = D,\ R,\ L)$
```

Fig. 1. Algorithm to compress $D = A'$ using $P = \Psi$ in $O(n)$ time. We use $next(j)$ as a shorthand for "**if** $L[j+1] = 1$ **then** $j+1$ **else** $D[j+1]$". Alternatively, we can replace $next(j)$ by $selectnext(L, j)$ and remove the lines marked "//invalidate".

We keep running passes over D (using P) as long as we replace at least $\alpha n'$ pairs in a pass, where $0 < \alpha < 1$ is a parameter and n' is the number of valid elements in D in the previous pass.

Figure 1 shows a more detailed pseudocode. Its main difference with our description is that, instead of keeping track of which is the k such that $i' = P[k]$ in order to set $P[k] \leftarrow P[P[k]]$ when invalidating i', we defer this action to the next time we visit i'. Only then we alter P to skip i', and the amortized complexity is the same.

Cost. Let n_i be the number of valid elements of D in the i-th pass, then $n_{i+1} \leq (1-\alpha)n_i$. Since $n_0 = n$, it holds $n_i \leq (1-\alpha)^i n$. The i-th pass costs $O(n_i)$ time. Let k be the number of passes doing more than $\alpha n'$ replacements (i.e., all but the last one). The total cost is at most

$$\sum_{i=0}^{k}(1-\alpha)^i n + (1-\alpha)^k n \ \leq\ n\sum_{i\geq 0}(1-\alpha)^i + (1-\alpha)^k n \ \leq\ \frac{1}{\alpha}\, n \ =\ O(n).$$

Thus our algorithm achieves linear time while requiring only the space for D (overwritten on A' and finally leaving there C), for P (overwritten on Ψ), and for L (which is also needed in the final structure). It is also simple and fast in practice.

3.4 Stronger Compression based on Ψ: Algorithm RPΨ

The only advantage of using the original Re-Pair is that it yields better compression and enforces the property that each new rule in the dictionary removes no more pairs than the previous rule. The latter comes from the fact that the pairs in Re-Pair are replaced in decreasing order of frequency. This prevents less frequent pairs to break longer chains of replacements. We now modify the algorithm that uses Ψ to obtain compression ratios as close to Re-Pair's as desired, at the expense of $O(n \lg n)$ time complexity (multiplied by a constant that increases as the compression ratio improves). The key idea is to replace longer chains first.

Algorithm. The algorithm is as follows:

—We make one pass searching for the longest chain of equal pairs obtained by following Ψ, let f be its length.

—We apply algorithm RPΨ_0, yet we only replace the chains of length at least $t_0 = \delta \cdot f$, where $0 < \delta < 1$ is a parameter.

—We apply algorithm RPΨ_0 successively, using $t_1 = \delta \cdot t_0$, then $t_2 = \delta \cdot t_1$, and so on, until $t_i \leq \gamma$, which is another parameter. At this point we decrement t_i one by one until we reach $t_i = 1$.

Cost. We already know that the total cost of all passes that replace more than $(1-\alpha)n'$ elements adds up to $O(n)$. The number of passes where we replace less than $(1-\alpha)n'$ pairs, on the other hand, is at most $\lg_\delta f + \gamma$. This is, $\lg_\delta f$ for the part where $t_{(\cdot)}$ decreases by a δ fraction, plus γ for the part where $t_{(\cdot)}$ decreases one by one. Thus the total cost is at most

$$\frac{1}{\alpha}\, n + (\lg_\delta f + \gamma)\, n.$$

Now, if we choose a constant s, $\alpha = 1/(s \cdot \lg n)$, and $\gamma = \lg n$, the total time is $O(n \lg n)$. Choosing other values of s, δ and γ we obtain better complexities, but worsen the compression quality. Within $O(n \lg n)$ complexity, we can improve the compression ratio by tuning constants δ and s.

3.5 Compressing the Dictionary

We now develop a technique to reduce the size of the representation of the dictionary of rules R. This can be of independent interest for Re-Pair in general. Good compression techniques for R do exist [Larsson and Moffat 2000] and perform much better than ours. However, they are not comparable. Ours is not just a *compressed representation* of R, but a *compressed data structure*, that is, it gives direct access to any rule of R without "decompressing" the representation. A compressed representation of rules is useful for decompressing the whole sequence, while a compressed data structure fits our purpose of accessing C at random positions.

We start with the observation that, if we have a rule $s \to ab$ and s is only mentioned in another rule $s' \to sc$, then we could perfectly remove rule $s \to ab$ and rewrite $s' \to abc$. This gives a net gain of one integer, but now we have rules of varying length. This is easy to manage, but we prefer to go further. We develop a technique that permits *eliminating every rule definition that is used within R, once or more, and gain one integer for each such rule eliminated.*

The key idea is to write down explicitly the topology of the binary tree formed by expanding the definitions (by doing a preorder traversal and writing 1 for internal nodes and 0 for leaves), so that not only the largest symbol (tree root) can be referenced later, but also any subtree.

Representation. Initially, each rule $s \rightarrow ab$ is seen as a small forest with one internal node (s) and two leaves (a and b). The described preorder traversal of such a small tree yields the binary representation `100`, and the sequence of leaves ab. The binary representations are concatenated in a bitmap R_B and the leaves in a sequence R_S. Nonterminals will be identified with the position of their `1` in R_B, and their representation finishes when, starting at their first `1` and scanning to the right, we see one more `0` than `1`s. The `0`s in R_B are in one-to-one correspondence with the positions in R_S.

For example, consider the rules $R = \{s \rightarrow ab,\ t \rightarrow sc,\ u \rightarrow ts\}$, and $C = tub$. We first represent the rules by the bitmap $R_B =$ `100100100` (where s corresponds to position 1, t to 4, and u to 7) and the sequence $R_S = ab1c41$ (we are using letters for the original symbols of A', to distinguish them from the bitmap positions that describe the nonterminals). We express C as $47b$. To expand, say, 4, we go to position 4 in R_B and compute $rank_0(R_B, 4) = 2$ (i.e., the number of zeros up to position 4, $rank_0(R_B, i) = i - rank(R_B, i)$). Thus the corresponding symbols in R_S start at position $2 + 1 = 3$. We extract one new symbol from R_S for each new zero we traverse in R_B, and stop when the number of zeros traversed exceeds the number of ones (this means we have completed the subtree traversal, as explained). This way we obtain the definition $1c$ for symbol 4.

More formally, let $R = \{s_1 \rightarrow a_1 b_1, s_2 \rightarrow a_2 b_2, \ldots, s_k \rightarrow a_k b_k\}$, where indeed $s_k = n + k$ (as $n = A'[1] = A[1]$ is the maximum value in A'). Thus, we write down R_B, R_S, and the new C as follows (note that positions in R_B are written in R_S shifted by n to distinguish them from the original symbols):

—$R_B = (100)^k$.

—$R_S = a_1 b_1 a_2 b_2 \ldots a_k b_k = r_1 r_2 r_3 \ldots r_{2k}$, except that if $r_i > n$ we set it to $r_i = n + 1 + 3(r_i - n - 1)$, so that they point to the 1's in R_B.

—$C = c_1 c_2 \ldots c_{n'}$ undergoes the same transformation: if $c_i > n$, we set it to $c_i = n + 1 + 3(c_i - n - 1)$.

Reduction. Let us now reduce the dictionary, in our example, by expanding the definition of s within t (even if s is used elsewhere). The new bitmap is $R_B =$ `11000100` (where $t = 1$, $s = 2$, and $u = 6$), the sequence is $R_S = abc12$, and $C = 16b$. Note that the subtree of t corresponds to topology `11000` and leaves abc, whereas the subtree of s is within t, with topology `100` and leaves ab. Although they share part of their representation, they can be referenced independently.

We can now remove the definition of t by expanding it within u. This produces the new bitmap $R_B =$ `1110000` (where $u = 1$, $t = 2$, $s = 3$), the sequence $R_S = abc3$ and $C = 21b$. Further reduction is not possible because u's definition is only used from C.[2] At the cost of storing at most $2|R|$ bits (for R_B), we can reduce R by one

[2]It is tempting to replace u in C, as it appears only once, but our example is artificial: A symbol that is not mentioned in R must appear at least twice in C.

```
**Algorithm** Compress_Dictionary($R = \{s_1 \to a_1b_1, \ldots, s_k \to a_kb_k\}$, $C = c_1 \ldots c_{n'}$)
    **for** $i \leftarrow 1 \ldots k$ **do** $U[i] \leftarrow 0$
    **for** $i \leftarrow 1 \ldots k$ **do**
        **if** $a_i > n$ **then** $U[a_i - n] \leftarrow 1$
        **if** $b_i > n$ **then** $U[b_i - n] \leftarrow 1$
    **for** $i \leftarrow 1 \ldots k$ **do** $NV[i] \leftarrow 0$
    $j \leftarrow 1$, $R_B \leftarrow \langle\rangle$, $R_S \leftarrow \langle\rangle$
    $LR_B \leftarrow 0$ // length in bits of bitmap $R_B$
    **for** $j \leftarrow 1 \ldots k$ **do**
        **if** $U[j] = 0$ **then** Expand_Rule($j$)
    **for** $i \leftarrow 1 \ldots n'$ **do**
        **if** $c_i > n$ **then** $c_i \leftarrow NV[c_i - n] + n$
    **return** $(R_B, R_S, C)$

**Algorithm** Expand_Rule($j$)
    $R_B \leftarrow R_B : 1$, $LR_B \leftarrow LR_B + 1$
    $NV[j] \leftarrow LR_B$
    **if** $a_j \leq n$ **then**
        $R_S \leftarrow R_S : a_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else if** $NV[a_j - n] > 0$ **then**
        $R_S \leftarrow R_S : NV[a_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else** Expand_Rule($a_j - n$)
    **if** $b_j \leq n$ **then**
        $R_S \leftarrow R_S : b_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else if** $NV[b_j - n] > 0$ **then**
        $R_S \leftarrow R_S : NV[b_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
    **else** Expand_Rule($b_j - n$)
```

Fig. 2. Algorithm to compress the dictionary R and to update C in $O(n)$ time. R_B, R_S, NV, and LR_B act as global variables, "$\langle\rangle$" is the empty sequence and ":" the concatenation operator.

integer for each definition that is used at least once within R.

The reduction can be easily implemented in linear time, avoiding the successive renamings of our example, as follows: We first check for each rule if it is used within R, marking this in a bitmap U. Then we traverse R and only write down (the bits of R_B and the sequence R_S for) the entries that are not used within R. We recursively expand those entries, appending the resulting tree structure to R_B and leaf identifiers to R_S. Whenever we find a created symbol that does not yet have an identifier, we give it as identifier the current position in R_B and recursively expand it. We store these new identifiers in an array NV. Otherwise the expansion finishes and we write down a leaf (a `"0"`) in R_B and the identifier in R_S. Then we rewrite C using the renamed identifiers. Figure 2 shows detailed pseudocode.

We can further compress the dictionary if we take into account that a rule only uses previous rules or original symbols. That is, the i-th rule can only point to elements with representation of length $\lceil \lg_2 i \rceil$ bits. With a simple arithmetic computation we can directly access any rule.

Another way to further compress the dictionary, yet with a time penalty, is as follows: Instead of using the position i of a rule inside bitmap R_B, use $j = rank_1(R_B, i)$. Given that j, we find the position in R_B where the rule starts with $i = select_1(R_B, j)$ (this is the inverse operation of $rank_1$, i.e., the position of the j-th 1 in R_B, and can also be computed in constant time using $o(|R_B|)$ bits [Munro

Text	n	σ	H_0	H_1	H_2	H_3	H_4	H_5
`dna`	100MB	16	1.977	1.932	1.922	1.918	1.911	1.902
`english`	100MB	239	4.556	3.630	2.949	2.417	2.047	1.807
`proteins`	100MB	27	4.190	4.168	4.150	4.073	3.835	3.042
`sources`	100MB	230	5.540	4.017	3.005	2.257	1.785	1.457
`xml`	100MB	97	5.228	3.336	2.070	1.374	0.996	0.770

Table II. Some characteristics of the text files we use.

1996; Clark 1996]). We gain at least 1 bit per rule in the dictionary and in the text.

4. EXPERIMENTAL RESULTS

In this section we first study the performance of our LCSA technique, and then compare it to other alternatives in the literature.

We use text collections obtained from the *PizzaChili* site.[3] This site offers a collection of texts of various types and sizes. We use the five types (`dna`, `english`, `proteins`, `sources`, and `xml`) for which 100MB files are available. Using larger datasets gives no additional clues on the performance.[4] Table II summarizes some of their properties. Some indexes will use more space when the alphabet size is large, while others will use less space when the k-th order entropy H_k decreases fast with k. Our dataset is interesting in this sense because `english`, `sources` and `xml` have large alphabet sizes (and relatively large H_0 entropies) and low high-order entropies. The low entropy of `dna`, instead, is only a consequence of its small alphabet size, and it barely decreases with the order considered.

The experiments were run on an Intel Core2 Duo, running at 3.0 GHz, with 6MB cache and 8GB RAM. The operating system was Linux 64-bit with kernel 2.6.24-31-server, and the compiler was `g++` version 4.2.4 with `-O3` optimization and `-m32` flag (as required by several packages tested).

Our data points are averages over 100,000 to 1,000,000 repetitions, sufficient in each case to ensure at most 5% of error with 95% confidence.

4.1 Compression Performance

Lacking freely available implementations of Re-Pair that worked properly on our datasets, we implemented ourselves such a compressor.[5]. It runs in linear time and requires $12n + O(p)$ bytes of main memory, where p is the number of distinct pairs generated at any time. We call RP this implementation in the sequel. Further, we call RPΨ the Ψ-based approximation that runs in $O(n \lg n)$ time (Section 3.4). We do not experiment directly with the simplified method of Section 3.3 because in practice it is a particular case of RPΨ. We also include the methods RPSP and RPΨSP, which correspond to the variants that limit the size of a nonterminal to $d = 256$ terminal symbols. In all cases, we take absolute A' samples each $l = 64$

[3]`http://pizzachili.dcc.uchile.cl`
[4]The LCSA can handle longer texts, but some indexes we compare with are older and they would not run on more than 100MB without significant reimplementation.
[5]Freely available at `http://www.dcc.uchile.cl/gnavarro/software` We used the "normal" version. There is a "balanced" version that performed almost exactly the same in all aspects, so it is omitted from the experiments we present here.

s	Bytes per cell `xml`	Bytes per cell `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1	1.0412	2.2188	96	306
2	1.0380	2.2140	113	328
4	1.0364	2.2128	135	360
8	1.0356	2.2120	155	408
16	1.0352	2.2116	168	450
32	1.0348	2.2112	190	538
64	1.0348	2.2212	206	622
128	1.0348	2.2112	215	714

Table III. Compression ratio obtained using different values of s for the approximation RPΨSP. In this case we use $\delta = 1/2$.

δ	Bits per cell `xml`	Bits per cell `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1/2	1.0356	2.2120	155	408
3/4	1.0324	2.2116	197	480
7/8	1.0296	2.2116	261	620
15/16	1.0272	2.2116	354	930
31/32	1.0240	2.2116	523	1,588

Table IV. Compression obtained using different values of δ using approximation RPΨSP. In this case we use $s = 8$.

positions. As we will see, the choice of d affects compression very little (at most 6%), whereas the chosen l value adds only 0.5 bits per cell, which adds 2%–7%. In exchange, they sharply speed up extraction, whereas using smaller values does not give further significant reductions.

To tune the parameters of the approximate variant RPΨSP, we test different values on two small files (`english` and `xml`, truncated to 50MB). We show, among several we carried out, the following experiments, as they best reflect the choice of parameters. Table III shows that the compression gain for increasing s loses importance for $s > 8$. Table IV, on the other hand, shows that increasing δ does not give any gain on `english`, yet it slightly improves compression ratio on `xml`. A fair choice of parameters for RPΨSP, which we use for the rest of the experiments, is $s = 8$, $\delta = 3/4$ and $\gamma = \lg n$.

Now that we have fixed a convenient parameterization for our compression algorithms, we compare them to each other on all the collections. Table V shows that the compression ratio varies widely. On `xml` data we achieve less than one byte per cell, whereas compression is extremely poor on `dna` (close to the 3.375 bytes per cell needed by a plain representation using $\lceil \lg n \rceil$ bits). In other text types of interest (`english`, `sources`) we slash the suffix array size to around a half.

Below the name of each text we write $H_k/H_0 \cdot (\lceil \lg n \rceil/8)$, for $k = 2$ or 3, which translates to bytes per cell the high-order statistical compressibility of the collection independently of its alphabet size. The measure turns out to be a good (slightly optimistic) predictor of the compression we attain, except for `proteins`, where even for $k = 4$ it is pessimistic (it becomes too optimistic for $k = 5$).

Note, on the other hand, that the Ψ-based method achieves compression only very slightly worse than that of Re-Pair. In addition, it is an order of magnitude slower to build. It requires, however, 1.5 to 4 times less construction space, which can make it more practical depending on the available main memory. The construction

Coll. H_k/H_0	Method	Bytes / cell	Compr. time (s)	Compr. space	Expected decompr.	Dict. compr.	Dict. vs $\vert A\vert$
`dna`	RP	3.36	114	40.14	5.18	79.86%	14.34%
3.28	RPSP	3.37	115	40.27	4.36	79.09%	14.28%
(H_2)	RPΨ	3.36	1,029	11.80	4.94	79.22%	14.39%
	RPΨSP	3.37	1,061	12.13	4.28	79.41%	14.34%
`english`	RP	2.27	132	32.73	242.67	58.98%	22.31%
2.18	RPSP	2.29	134	32.49	32.12	59.38%	22.29%
(H_2)	RPΨ	2.28	1,010	12.52	80.10	59.22%	22.42%
	RPΨSP	2.30	982	12.21	29.99	59.60%	22.41%
`proteins`	RP	2.85	105	37.57	59.72	79.47%	4.72%
3.09	RPSP	2.86	105	37.76	13.91	79.82%	4.74%
(H_4)	RPΨ	2.86	1,030	11.75	38.03	78.93%	5.03%
	RPΨSP	2.88	1,014	12.79	11.92	78.80%	5.12%
`sources`	RP	1.55	115	23.81	2,031.77	57.45%	15.60%
1.37	RPSP	1.59	119	24.04	62.38	58.09%	15.56%
(H_3)	RPΨ	1.57	700	13.01	260.04	57.56%	15.87%
	RPΨSP	1.61	1,016	13.00	56.89	58.13%	16.01%
`xml`	RP	0.93	80	18.61	6,796.00	57.10%	8.08%
0.89	RPSP	0.99	85	18.86	136.21	58.11%	8.44%
(H_3)	RPΨ	0.96	476	11.95	675.88	57.31%	8.31%
	RPΨSP	1.03	488	12.97	95.83	58.23%	8.87%

Table V. Structure size and build time using Re-Pair and its Ψ-based approximation. We also include versions with rules up to length $d = 256$ (SP extension). Compression space is measured in number of times the text size.

time does not include that to build the suffix array, which is the same (around 100 seconds) in all the methods and texts.

Table V also gives other statistics. Column 6 measures the average length of a cell of C if we choose uniformly in A (longer cells are in addition more likely to be chosen for decompression). Those values are proportional to the cost of decompressing a random cell, and illustrate the relevance of limiting the size of the rules to d, especially when compression is better (`sources`, `english`). Note that the values are related to compressibility, but not as much as one could expect. Rather, they owe to a more detailed structure of the suffix array: they are higher when the compression is not uniform across the array. In every case, we can limit the maximum length of a C cell. The SP variants, where we force $d = 256$, show how such a limitation drastically reduces the access time while it impacts very little in compression ratio: at most by 5%, and generally much less. Note also that the Ψ-based variant, even if not restricted in length, produces shorter nonterminals than the Re-Pair based variant.

Column 7 shows the compression ratio achieved on the dictionary part using the technique of Section 3.5, charging it the bitmap introduced as well. It can be seen that the technique is rather effective, approaching a compression ratio below 60% on compressible texts and below 80% on incompressible ones. We remind that 50% is the best possible ratio our dictionary compression technique can reach. (The compression ratios of previous columns do account for the dictionary space and all the necessary structures to operate.)

The last column shows the fraction of the original suffix array size (measured as $n\lceil\lg n\rceil$ bits) required by the dictionary. This is the part of the data where random accesses are carried out at decompression time, whereas the other accesses are local, in the compressed sequence. The rather low percentages show that the structure is cache-friendly, so high extraction speeds can be expected.

4.2 Comparison with the State of the Art

In this section we compare our LCSA against various alternative suffix array representations. Various of the compressed suffix arrays we wish to compare with [Mäkinen 2003; Sadakane 2003; Ferragina and Manzini 2005; Mäkinen et al. 2010] already have competitive implementations, and we have just slightly adapted them. There are two important data structures, however, that existed only as theoretical proposals [Grossi and Vitter 2005; Rao 2002]. In the Appendixes we detail how we have implemented and optimized them, and study their best parameterization.

LCSA. Our LCSA, considering variants $LCSA$ = RPSP and $LCSA\Psi$ = RPΨSP, with the parameters set as above, and using samplings $l = 64$ and $d = 256$.

MakCSA. The Compact Suffix Array of Mäkinen [2003], implemented by himself.[6] In Appendix B we study its best parameterization for this problem. The code can only search for patterns and list their positions. In order to extract arbitrary ranges of A we added a compressed bitmap [Raman et al. 2007] of length n, marking the starting positions of the blocks, so that we could convert positions in A to positions in the compacted array.

GVCSA. The Compressed Suffix Array of Grossi and Vitter [2005], implemented by ourselves. See the details in Appendix C.

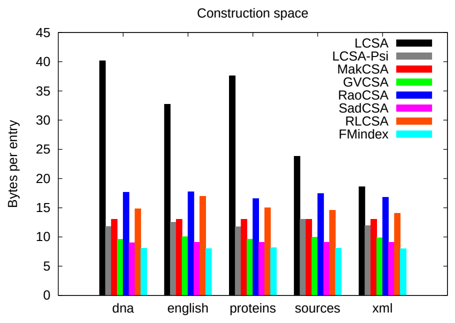
RaoCSA. The structure of Rao [2002], implemented by ourselves. See the details in Appendix D.

SadCSA. The Compressed Suffix Array of Sadakane [2003], implemented by himself (the implementation is available at *PizzaChili*).[7] This has two parameters: s_Ψ, the sampling step to access the compressed Ψ array, which is left at $s_\Psi = 128$, where it performs best, and s_A, the sampling step to store samples of A, which is used as the space/time tradeoff parameter. We consider values $s_A = \{4, 8, 16, 32, 64, 128, 256\}$. We used routines tailored to extract various consecutive cells, taking advantage of runs of consecutive Ψ values, that were already in Sadakane's code.

RLCSA. The Run-Length Compressed Suffix Array of Mäkinen et al. [2010], implemented by Jouni Sirén (the implementation is available at *PizzaChili*).[8] The *RLCSA* is a variant of *SadCSA* specialized on handling repetitive texts. It has the same parameters s_Ψ (which we use at its default value 32) and s_A, which we use as the space/time tradeoff parameter, considering values $s_A = \{4, 8, 16, 32, 64, 128, 256\}$. *RLCSA* also has routines tailored to extract various consecutive cells.

FMindex. The FM-index [Ferragina and Manzini 2005], in its most recent and efficient variant [Kärkkäinen and Puglisi 2011], implemented by themselves.[9] We used the suffix array sampling parameters $s_A = \{4, 8, 16, 32, 64, 128, 256\}$, and the text sampling parameter set to infinity. We only show the variant using plain bitmaps, as the time/space obtained with compressed bitmaps were almost identical in this scenario.

[6] Downloaded from `http://www.cs.helsinki.fi/u/vmakinen/software/csa.zip`
[7] At `http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array`
[8] At `http://pizzachili.dcc.uchile.cl/indexes/RLCSA/`
[9] Thanks to Simon Puglisi for handing us the code.



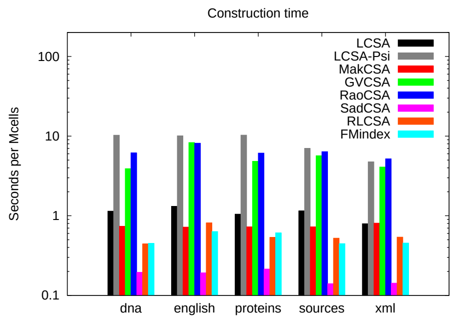


Fig. 3. Construction time and space for the different indexes on each text.

We remind that *SadCSA*, *RLCSA* and *FMindex* are self-indexes, but we are not interested in this feature for this experiment. We only evaluate their ability to reproduce a range of suffix array cells.

Figure 3 compares construction space and time for all the indexes. We have not considered the space and time to build the suffix array, as this is orthogonal to the index construction problem and must be done for all the indexes. It can be seen that all construction spaces are relatively close, except that of *LCSA*, which in bad cases can require as much as 40 bytes per entry. Those requiring the least space, around 7–9 bytes per entry, are *GVCSA*, *SadCSA*, and *FMindex*. The others are usually within 12 bytes per entry, except *RaoCSA* and *RLCSA*, which may require up to 16–17 bytes per entry.

With respect to construction time, *SadCSA* again excells, requiring less than 0.2 seconds per million cells, whereas the next faster indexes (*FMindex*, *RLCSA*, *MakCSA*, and *LCSA*, more or less in that order) build at a rate of around 1 second per million cells. The other indexes build ten times slower.

LCSA builds fast (at about 1 second per million cells) but it may require too much extra space (up to 40 times the text size). Variant *LCSA*Ψ, although slower to build (about 10 seconds per million cells, which is still affordable even for large texts), requires reasonable space (near 12 times the text size, not far from the state of the art). We will show both variants in the experiments that follow.

Figure 4 shows the space/time tradeoffs, for all the indexes on all the texts, to access a random cell. The space is shown as the index size in bytes divided by n, that is, in bytes per cell.

It can be seen that *SadCSA* and *FMindex* are the clear winners in all cases, being faster and smaller than all the others. The size of these indexes is sensitive to the high-order entropy of the texts, whereas *GVCSA* and *RaoCSA* are more dependent on the alphabet size. Among the two, *GVCSA* is always better than *RaoCSA*. *RLCSA*, instead, is sensitive to the repetitiveness of the text, performing worst on `dna` and best on `xml`. Finally, in both *MakCSA* and the variants of our *LCSA* the space depends more on the relation between the high and the zero order entropies of the texts, H_k/H_0. Thus, they perform particularly bad on `dna` and `proteins`, much better on `english` and `sources`, and particularly well on `xml`. Yet, they are still slower than *SadCSA* and *FMindex*.

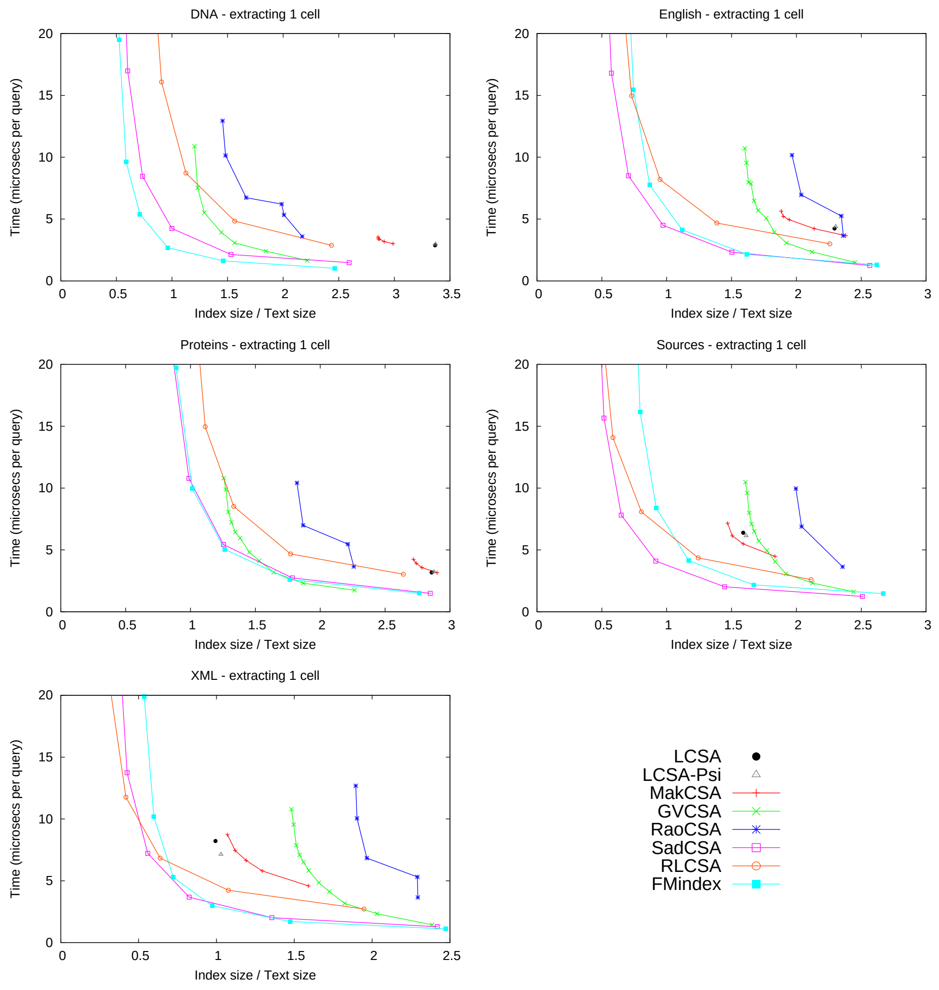


Fig. 4. Time/space tradeoffs to access one random cell.

The relation between *MakCSA* and *LCSA* variants is mixed. In cases like `dna` and `english`, the former performs better in time and space. On `proteins` and `sources`, *LCSA* competes in space, but the time is either equal or dominated by that of *MakCSA*. Finally, on `xml`, where both perform best in space, the *LCSA* variants use less space than *MakCSA*, and dominate it in time too.

The situation changes when we consider the performance to access a number of consecutive cells, which is the case, for example, when reporting all the occurrences of a pattern in the text. Figures 5 and 6 show the space/time tradeoffs when retrieving 10 and 100 consecutive cells, respectively. *RLCSA* becomes relatively faster, matching *SadCSA* on `sources` and outperforming it on `xml`. However, *LCSA* and *MakCSA* improve much faster. Already for extracting as few as 10 consecutive cells, the *LCSA* variants and *MakCSA* become competitive with the fastest alternatives,

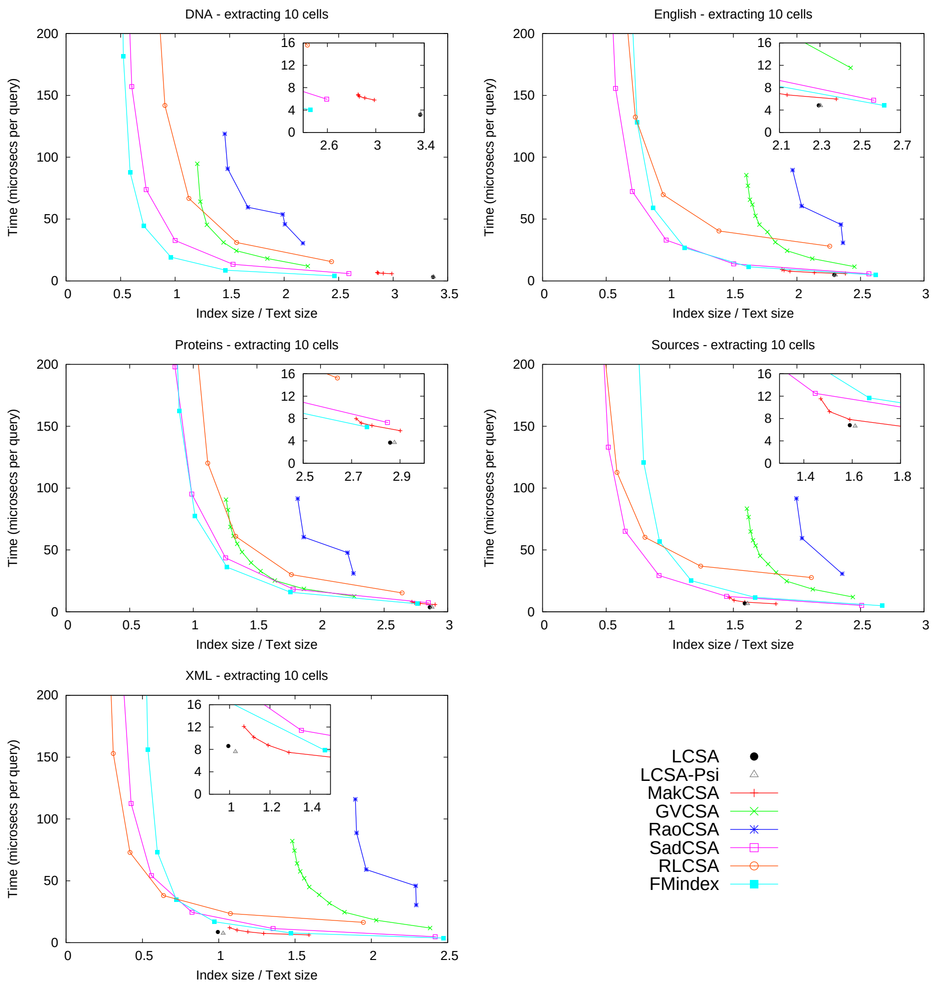


Fig. 5. Time/space tradeoffs to access 10 random cells.

and when extracting 100 consecutive cells they become an order of magnitude faster. This is the scenario where *LCSA* and *MakCSA* excell. The (zoomed) plots also show that *LCSA* becomes faster than *MakCSA* in all cases (for the same space) as soon as we access various consecutive cells. We study this aspect in more detail next.

Figure 7 shows, for *SadCSA*, *FMindex*, *MakCSA*, and *LCSA* variants, the time to extract 1 to 20 consecutive cells starting at a random position. We chose the parameterizations that gave the others the best time without using significantly more space than that of *LCSA*. We left aside `dna`, since *LCSA* does not obtain any significant compression on it, and thus the "fair comparison in space" should be with a plain suffix array. In some cases the self-indexes look better than in others because they compress better and thus can use a denser sampling for the same space. It can be seen that, as soon as we extract a few consecutive cells,

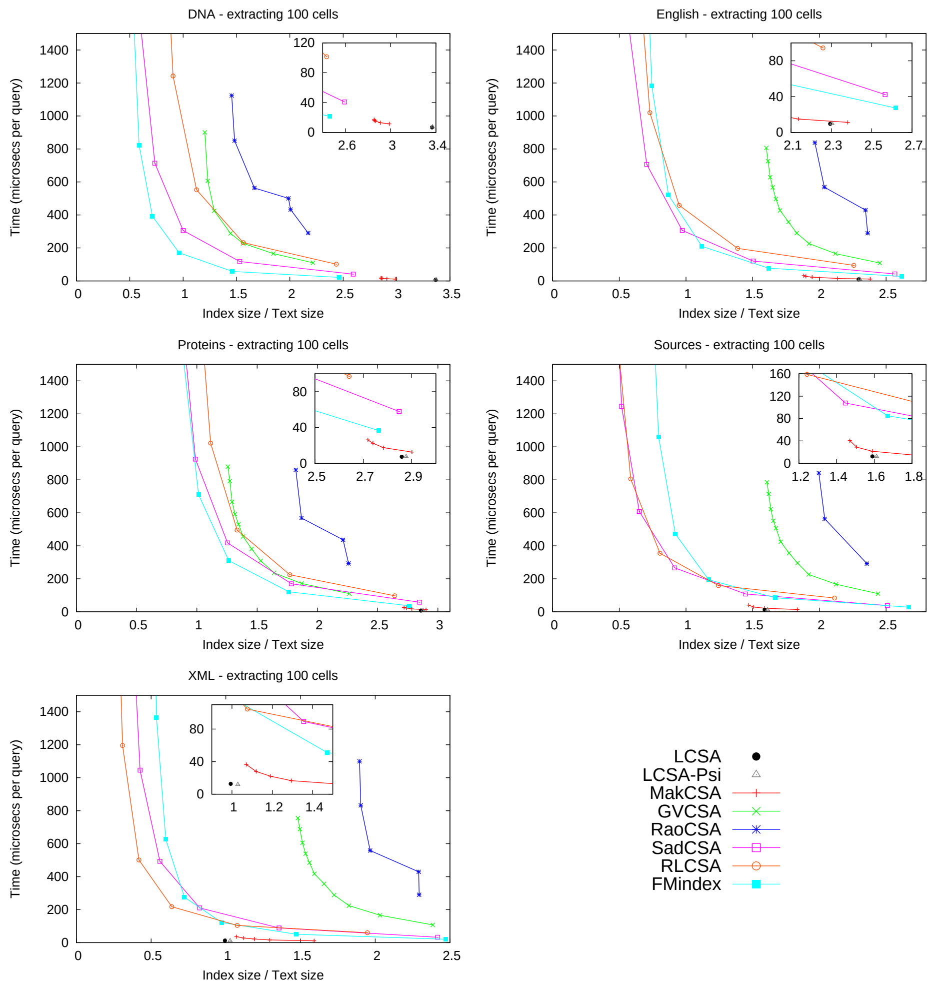


Fig. 6. Time/space tradeoffs to access 100 random cells.

SadCSA and *FMindex* become the slowest alternatives (having been the fastest for extracting one isolated cell), due to their direct linear increase in time. On the other hand, the *LCSA* variants are always faster than *MakCSA*, and in the most compressible texts (`sources` and `xml`) it is clear that the growth rate of the time is also lower on *LCSA*.[10]

Finally, we tested a plain suffix array implementation, where each cell uses $\lceil \lg n \rceil$ bits (i.e., 3.375 bytes per cell in our texts). The access time for one cell is around

[10]It can be noted that the time to extract an individual cell in *MakCSA* is closer to that of *LCSA* in Figure 4 than in this plot. This is because some optimizations are possible when extracting just one cell. Those could laboriously be extended to extract more (with specific code for each number of cells), but this would have impact only to extract very few cells. The times would rapidly converge to those of the general algorithm.

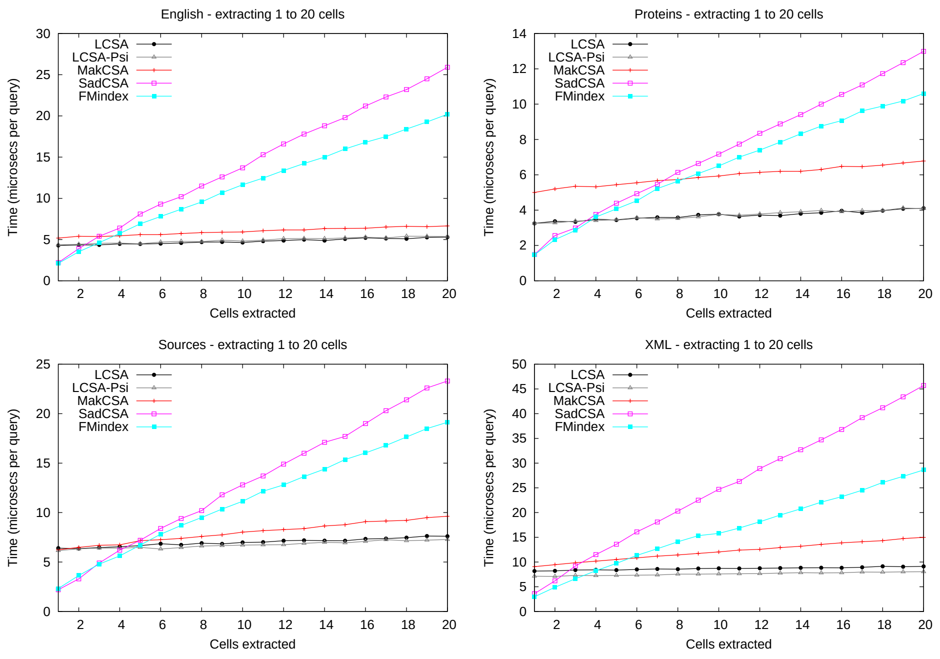


Fig. 7. Time to access 1 to 20 consecutive cells, using about the same space. *SadCSA* and *FMindex* use sampling values 4, 2, 4, and 8 for the plots in reading order. For *MakCSA* we used best C for each text, as before, and then, to obtain about the same space, $D = 3$ on `english`, $D = 5$ on `proteins`, $D = 10$ on `sources`, and $D = 30$ on `xml`.

250 nanoseconds (note it might be necessary to access two consecutive memory words to extract one value), and then increases at a rate of around 6 nanoseconds per consecutive cell accessed. This is about 10 times faster than *LCSA*.

Our *LCSA* improves its compression effectiveness on repetitive collections, as is the case of `sources` and, more clearly, `xml`. It is tempting to test it in the scenario of highly repetitive collections [Mäkinen et al. 2010]. However, *RLCSA* exploits much better those collections, while the improvement of *LCSA* is not much better than that of *SadCSA*. For example, on the S. pardoxus collection [Mäkinen et al. 2010], the *RLCSA* obtains acceptable performance (near 15 microseconds to compute a single cell value) using 0.15 times the text size, whereas *SadCSA* uses 0.27 times the text size. *LCSA* uses at least 0.24 times the text size, which is much better than its performance on the general collections considered in this paper, but not significantly better than *SadCSA*, and clearly worse than *RLCSA*.

5. CONCLUSIONS AND FUTURE WORK

We have presented a suffix array compression method, *Locally Compressed Suffix Arrays (LCSA)* that retains fast access to suffix array cells, especially when a number of contiguous cells must be read. We have proved analytically and experimentally that the size of the LCSA is related to the k-th order entropy of the text, performing particularly well on texts where their k-th order entropy is significantly smaller than the zero-order entropy (structured XML collection and source code repositories are good examples, although the space performance is also decent on plain English text collections). On the other hand, compression is not good on DNA or Protein sequences. Extracting c consecutive cells is proven to take the almost-optimal time $O(c) + o(\lg n)$, and to be very fast in practice: the LCSA is the fastest to extract $c = 7$ contiguous cells or more, thanks to its local decompression properties. The LCSA is a viable alternative to classical suffix arrays, as well as to current self-indexes, which use less space but are much slower to extract suffix array intervals. An implementation has been left publicly available at `http://pizzachili.dcc.uchile.cl/additionalSuffixArrays.html`.

As a byproduct, we have presented Re-Pair algorithms tailored to suffix array differences, which exploit the structure of Ψ to run using much less memory than the general algorithm. Those new algorithms are approximations, yet we show that their compression loss is negligible.

Another byproduct, which might be of general interest, is a compact data structure to represent the Re-Pair dictionary. This structure can reduce the dictionary space by up to 50%, and operates in compressed form, that is, it permits decompressing parts of the text without uncompressing the dictionary. In practice the dictionary compression achieved is closer to 60%.

Since its conference publication [González and Navarro 2007], the key ideas of this work have been shown to apply in other scenarios. In document retrieval, where the goal is to list the documents where the pattern appears (as opposed to all of its exact positions) a successful approach is to use a "document array" $D[1, n]$, which records in $D[i]$ the document to which $A[i]$ belongs. As the (shifted) repetitions in A translate into exact repetitions in D (except at d positions, where d is the number of distinct documents in a collection), Gagie et al. [2013] used our results to give the first scheme to compress D, and Navarro et al. [2011] showed its practicality, achieving the only technique so far to compress D. Another application is compressed suffix trees, where in addition to $A[1, n]$ one needs to encode $LCP[1, n]$, which gives the length of the longest common prefix between consecutive suffixes. While there is a technique to compress $LCP[1, n]$ into just $2n + o(n)$ bits [Sadakane 2007], Fischer et al. [2009] showed that it could be compressed into $o(n)$ bits on sufficiently compressible collections. For this sake, they used and extended the results we proved in this work, showing that shifted repetitions also appear on the LCP array. Abeliuk et al. [2013] implemented this idea on highly repetitive collections, and also applied our ideas more directly by representing the LCP in differential form and applying Re-Pair on it. Finally, the technique to compress the Re-Pair dictionary has been used a number of times in other practical scenarios where Re-Pair compression was used [Claude and Navarro 2010b; 2010a; Claude et al. 2010; 2011; Navarro et al. 2011].

Our work leaves several future development lines. An important one is to take advantage of the locality of the LCSA in order to deploy it on disk. If the dictionary of rules can be maintained in main memory (in our examples it took 5–22% of the suffix array size, and we can limit its size if desired), then the access to the compressed sequence is local, and disk I/Os will furthermore be reduced thanks to compression. We are focusing on merging the compressed suffix tree presented by

Clark and Munro [1996] with our structure. We believe the result would be quite competitive in practice.

In the longer term, we believe this is a relevant step towards compressed text indexes with competitive locating times, in particular via locality of access. The key was to build on the runs in Ψ, which have been used in the past to achieve compression, yet not access locality. We showed that the regularities that Re-Pair exploits on the differential suffix array are closely related to those runs in Ψ. Thus we could take advantage of the locality properties of Re-Pair, and also used the close relation with Ψ to analyze the compression achieved and design faster Re-Pair variants for this case. Our resulting index is still far from achieving the space used by the smallest self-indexes, which are however extremely slow to locate. Is there a fundamental lower bound to the tradeoff one can achieve between space and time for locating? Is there a limit to what can be achieved via local compression? Pursuing this challenge is probably one of the most exciting research directions nowadays in compressed text indexing.

REFERENCES

Abeliuk, A., Cánovas, R., and Navarro, G. 2013. Practical compressed suffix trees. *Algorithms 6,* 2, 319–351.

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2,* 1, 53–86.

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

Baeza-Yates, R. and Gonnet, G. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM 43,* 6, 915–936.

Belazzougui, D. and Navarro, G. 2011. Alphabet-independent compressed text indexing. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*. LNCS 6942. 748–759.

Clark, D. 1996. Compact pat trees. Ph.D. thesis, University of Waterloo, Canada.

Clark, D. and Munro, I. 1996. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 383–391.

Claude, F., Fariña, A., Martínez-Prieto, M., and Navarro, G. 2010. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*. 86–91.

Claude, F., Fariña, A., Martínez-Prieto, M., and Navarro, G. 2011. Indexes for highly repetitive document collections. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*. 463–468.

Claude, F. and Navarro, G. 2010a. Extended compact web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, T. Elomaa, H. Mannila, and P. Orponen, Eds. LNCS 6060. Springer, 77–91.

Claude, F. and Navarro, G. 2010b. Fast and compact web graph representations. *ACM Transactions on the Web (TWEB) 4,* 4, article 16.

Claude, F., Navarro, G., Peltola, H., Salmela, L., and Tarhio, J. 2012. String matching with alphabet sampling. *Journal of Discrete Algorithms 11*, 37–50.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology.* World Scientific.

Elias, P. 1974. Efficient storage and retrieval by content and address of static files. *Journal of the ACM 21*, 246–260.

Fano, R. 1971. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts.

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

FERRAGINA, P. AND MANZINI, G. 2000. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*. 390–398.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed texts. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3,* 2, article 20.

FISCHER, J. 2010. Wee LCP. *Information Processing Letters 110*, 317–320.

FISCHER, J., MÄKINEN, V., AND NAVARRO, G. 2009. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science 410,* 51, 5354–5364.

GAGIE, T., KÄRKKÄINEN, J., NAVARRO, G., AND PUGLISI, S. 2013. Colored range queries and document retrieval. *Theoretical Computer Science 483*, 36–50.

GONNET, G., BAEZA-YATES, R., AND SNIDER, T. 1992. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Chapter 3: New indices for text: Pat trees and Pat arrays, 66–82.

GONZÁLEZ, R. AND NAVARRO, G. 2007. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4580. 216–227.

GROSSI, R., GUPTA, A., AND VITTER, J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

GROSSI, R. AND VITTER, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*. 397–406.

GROSSI, R. AND VITTER, J. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press.

KÄRKKÄINEN, J. 1995. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 937. 191–204.

KÄRKKÄINEN, J. AND PUGLISI, S. J. 2011. Fixed block compression boosting in FM-indexes. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*. 174–184.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996a. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*. Carleton University Press, 141–155.

KÄRKKÄINEN, J. AND UKKONEN, E. 1996b. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON)*. LNCS 1090. 219–230.

KURTZ, S. 1998. Reducing the space requirements of suffix trees. Report 98-03, Technische Kakultät, Univ. Bielefeld, Germany.

LARSSON, J. AND MOFFAT, A. 2000. Off-line dictionary-based compression. *Proc. IEEE 88,* 11, 1722–1732.

MÄKINEN, V. 2000. Compact suffix array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 1848. 305–319.

MÄKINEN, V. 2003. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae 56,* 1–2, 191–210.

MÄKINEN, V. AND NAVARRO, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

MÄKINEN, V., NAVARRO, G., SIRÉN, J., AND VÄLIMÄKI, N. 2010. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology 17,* 3, 281–308.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing 22,* 5, 935–948.

MANZINI, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

MCCREIGHT, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23,* 2, 262–272.

Munro, I. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 1180. 37–42.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

Navarro, G., Puglisi, S., and Valenzuela, D. 2011. Practical compressed document retrieval. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*. LNCS 6630. 193–205.

Okanohara, D. and Sadakane, K. 2007. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 60–70.

Raman, R., Raman, V., and Rao, S. S. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms 3,* 4, article 43.

Rao, S. 2002. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters 82,* 6, 307–311.

Russo, L., Navarro, G., and Oliveira, A. 2011. Fully-compressed suffix trees. *ACM Transactions on Algorithms 7,* 4, article 53.

Sadakane, K. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Symposium on Algorithms and Computation (ISAAC)*. LNCS 1969. 410–421.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms 48,* 2, 294–313.

Sadakane, K. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems 41,* 4, 589–607.

Ukkonen, E. 1993. Approximate string matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 228–242.

Weiner, P. 1973. Linear pattern matching algorithm. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*. 1–11.

A. ANALYSIS OF LCSA SPACE AND TIME

We now analyze the compression ratio of our data structure, first when using Re-Pair compression and then when using our approximate methods based on Ψ. We start with a lemma that limits the number of distinct pairs in A'.

Lemma 3. *There are at most 2ρ different pairs in A'.*

Proof. Note that, except for the first cell of each run, it holds $A'[i] = A'[\Psi(i)]$ within the run. Thus, if we cut off the first cell of each run, obtaining up to 2ρ runs in A', it holds that every pair $A'[i]A'[i+1]$ contained in such runs must be equal to $A'[\Psi(i)]A'[\Psi(i)+1]$. Therefore, the only pairs of cells $A'[i]A'[i+1]$ that are not equal to $A'[\Psi(i)]A'[\Psi(i)+1]$ are those where i is the last cell of its run. This shows that there are at most 2ρ different pairs in A', as a traversal following Ψ permutation will change pairs at most 2ρ times. $\square$

Re-Pair Compression. Since there are at most 2ρ different pairs, the most frequent pair appears at least $\frac{n}{2\rho}$ times. Because of overlaps, it could be that only each other occurrence can be replaced. Therefore, the total number of replacements in the first iteration is at least βn, for $\beta = \frac{1}{4\rho}$.

After we choose and replace the most frequent pair, we end up with at most $n(1-\beta)$ integers in A'. The number of runs has not increased, because a replacement cannot split a run. Thus, the same argument shows that the second time we end up with at most $n(1-\beta)^2$ symbols, and so on.

After M iterations, the length of C is at most $n(1-\beta)^M$ and R contains M rules. Assume for a while $\rho/n \leq 1/4$, thus $\beta n \geq 1$. Our upper bound to the total size, $|C| + 2|R| \leq n(1-\beta)^M + 2M$, is optimized for $M^* = \frac{\ln n + \ln\ln \frac{1}{1-\beta} - \ln 2}{\ln \frac{1}{1-\beta}}$, where it is $\frac{2(\ln n + \ln\ln \frac{1}{1-\beta} - \ln 2 + 1)}{\ln \frac{1}{1-\beta}}$. This is an upper bound to the final size of Re-Pair because it upper bounds the size after M^* iterations and, since Re-Pair shortens the total file size at each new iteration, the final size cannot be worse than that after M^* iterations. Since $\ln \frac{1}{1-\beta} = \ln \frac{4\rho}{4\rho-1} = \frac{1}{4\rho}(1+O(\frac{1}{\rho}))$, we have that $M^* = 4\rho \ln \frac{n}{\rho} + O(\rho)$ and that the total size is at most $8\rho \ln \frac{n}{\rho} + O(\rho)$ integers. Even if M grows up to n, the nonterminals can be represented using $O(\lg n)$ bits, thus the total space achieved is $O(\rho(1 + \lg \frac{n}{\rho}) \lg n)$ bits.

LEMMA 4. *Re-Pair compression reduces the size of the differential suffix array A' to $O(\rho(1 + \lg \frac{n}{\rho}) \lg n)$ bits, where ρ is the number of runs in A.*

As a comparison, the Compact Suffix Array of Mäkinen [2003] needs $O(\rho \lg n)$ bits of space [Navarro and Mäkinen 2007], which is asymptotically better. Our experiments confirm that this structure can be slightly smaller, yet at the price of being significantly slower.

$RP\Psi_0$ Compression. We now show that the simplified replacement method of Section 3.3 reaches the same asymptotic space complexity.

Just as for Re-Pair, the traversal using Ψ will create up to 2ρ pairs per pass. Assume for simplicity that, as we find each new pair in the traversal using Ψ, we always replace the pair, even if this involves creating it in R for just one occurrence in C (this is never better than the real algorithm). Thus we try to make all the $|A'|$ replacements, but we may fail because replacements overlap. That is, assume we have *abcd* and first replace $s \rightarrow bc$. In the new sequence *asd* we cannot make a replacement $s' \rightarrow ab$ nor $s' \rightarrow cd$. Indeed, in the best case we can carry out $\lfloor |A'|/2 \rfloor$ replacements, whereas in the worst case this is only $\lfloor |A'|/3 \rfloor$ (when we first choose all multiples of 3 as initial pair positions).

This shows that, in the first pass over Ψ, we add up to 4ρ integers (i.e., 2ρ rules) to R and remove at least $n/3$ integers from A'. For the next passes, since the number of runs is always limited by 2ρ, we can analyze the process using recurrence $S(n) = 4\rho + S(2n/3)$. If we stop the process after i iterations, we get $|C| + 2|R| = S(n) = 4\rho i + (2/3)^i n$, which is optimized for $i^* = \lg_{3/2}(n/\rho) + O(1)$ iterations, where we get $S(n) = 4\rho \lg_{3/2} \lg \frac{n}{\rho} + O(\rho) \leq 9.87\rho \lg \frac{n}{\rho} + O(\rho)$ integers.

$RP\Psi$ Compression. This analysis is similar to that of Re-Pair. The relevant invariant, which is easy to check from the description in Section 3.4, is as follows: *The approximate algorithm always replaces a pair that appears at least $\delta \cdot f$ times, being f the frequency of the currently most frequent pair.* In this sense, the algorithm acts as a δ-approximation to Re-Pair.

In Re-Pair, we replace first the most frequent pair, which appears at least $\frac{n}{2\rho}$ times. In RPΨ, we replace first a pair that appears at least $\frac{\delta n}{2\rho}$ times. This gives us a total number of replacements in the first iteration of at least $\beta' n$, where $\beta' = \delta\beta = \frac{\delta}{4\rho}$. The same occurs at each stage of the algorithm. Applying the same arguments of the analysis of Re-Pair with this new β', and the fact that $0 < \delta < 1$

is a constant, we obtain the same result as in Lemma 4.

LEMMA 5. *$RP\Psi_0$ and $RP\Psi$ compression reduce the size of the differential suffix array A' to $O(\rho(1+\lg\frac{n}{\rho})\lg n)$ bits, where ρ is the number of runs in A.*

The Final Space. We complete now the missing pieces to have a functional LCSA. First, recall that in Section 3.2 we introduced n/d unique symbols in A' to ensure that no nonterminal expands to more than d symbols, so that the extraction time could be bounded. This has the concrete effect, in the analysis, of increasing the number of runs by n/d, as each spurious symbol may cut one of the 2ρ runs. Then the total number of integers becomes $O(\rho\lg\frac{n}{\rho+n/d}+\frac{n}{d}\lg\frac{n}{\rho+n/d}+\rho+\frac{n}{d}) = O(\rho\lg\frac{n}{\rho}+\frac{n\lg d}{d}+\rho)$. By choosing $d=\lg^{\epsilon} n\lg\lg n$, for any $\epsilon>0$ (and $l=\lg^{\epsilon} n$, recall Section 3.2), we obtain the final result.

THEOREM 1. *The LCSA of a suffix array $A[1,n]$ with ρ runs occupies $O(\rho(1+\lg\frac{n}{\rho})\lg n+n\lg^{1-\epsilon} n)$ bits of space, for any $\epsilon>0$. It extracts any c consecutive cells of A in time $O(c+\lg^{\epsilon} n\lg\lg n)$.*

Time Performance. We describe how to use the LCSA for the typical suffix array tasks of counting and locating. This can be made more cleverly than plugging the structure blindly into the classic suffix array algorithms.

The absolute samples that are stored every l positions of A (i.e., array S) can be used to start the binary search. After $O(m\lg n)$ time, we can locate the area (of length l) where the binary search will terminate. This area can then be decompressed in time $O(l+\lg^{\epsilon} n\lg\lg n)$ according to Theorem 1, and then the binary search can terminate still within $O(m\lg n)$ time. Therefore counting time is not slowed down, asymptotically, due to our compression for any $\epsilon<1$ and $l=O(\lg n)$. Once we have identified the area $A[sp,ep]$ where the $occ=ep-sp+1$ occurrences lie, we can extract them all using Theorem 1, in time $O(occ+\lg^{\epsilon} n\lg\lg n)$. The two activities take time $O(m\lg n+occ)$, just as with a plain suffix array.

The time can be further improved by noting that the $O(m+\lg n)$ technique of Manber and Myers [1993] works equally well on an arbitrary set of strings, and thus we could use $O(n\lg^{1-\epsilon} n)$ additional bits of space to store longest common prefix information between the sampled suffixes of S, over which the binary search is carried out. The search within the decompressed block between two samples still needs to be done in the conventional way, in time $O(m\lg l)$. This yields an improved counting time of $O(m+\lg n+l+\lg^{\epsilon} n\lg\lg n+m\lg l)$. By choosing $l=\lg^{\epsilon} n$ we get the improved result.

THEOREM 2. *The LCSA of a suffix array A of a text $T[1,n]$ with ρ runs occupies $O(\rho(1+\lg\frac{n}{\rho})\lg n+n\lg^{1-\epsilon} n)$ bits of space, for any $0<\epsilon\le 1$. It counts the number of occurrences of any pattern $P[1,m]$ in time $O(m\lg\lg n+\lg n+\lg^{\epsilon} n\lg\lg n+occ)$. Two interesting particular cases are $O(\rho(1+\lg\frac{n}{\rho})\lg n+n)$ bits of space and $O((m+\lg n)\lg\lg n+occ)$ time ($\epsilon=1$), and $O(\rho(1+\lg\frac{n}{\rho})\lg n+n\lg\lg n)$ bits of space and $O(m\lg\lg n+\lg n+occ)$ time ($\epsilon=\frac{\lg\lg n-\lg\lg\lg n}{\lg n}$).*

B. TUNING THE SUFFIX ARRAY OF MÄKINEN

The suffix array of Mäkinen [2000; 2003] builds on the idea of detecting pairs of areas in A where the values differ by one unit, and encode one area by a pointer to

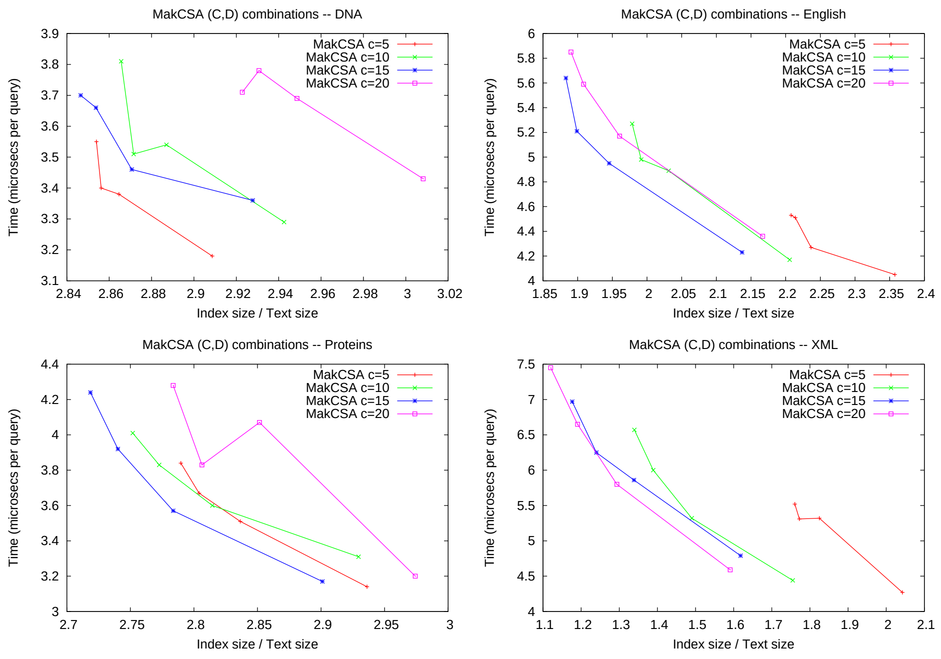


Fig. 8. Space/time tradeoffs to extract one cell using various options of parameters (C, D) for *MakCSA*, on four texts. We show one line per C value in $\{5, 10, 15, 20\}$, and one point in that line for each D value in $\{5, 10, 15, 20\}$ (smaller D uses more space and less time).

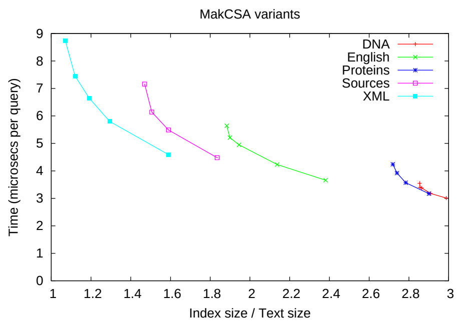
the other (note our LCSA exploits, in a different way, those same pairs of areas). The implementation of *MakCSA* (by its author) has two parameters, C and D, that control the maximum length of the areas and the maximum length of a referencing chain (since the pointed area may in turn be encoded as a pointer to a third area, slowing down extraction).

Figure 8 shows, for four texts, the space/time tradeoffs for accessing a random cell of A. It can be seen that the best values for C and D vary depending on the text. It turns out that $C = 5$ is the best for `dna`, $C = 20$ is the best for `xml`, and $C = 15$ for the rest, with some slight exceptions.

Figure 9 shows the performance over all the texts, using the best configuration. It can be seen that the space varies widely depending on the high-order compressibility of the texts, unlike other schemes that are blind to this aspect. On the right, where we show the time for extracting 100 consecutive cells, it can be seen that this structure performs much better in this scenario. That is, the times grow by a factor of around 6 instead of 100. Those configurations will be used for the main experiments to represent *MakCSA*.

C. IMPLEMENTING THE SUFFIX ARRAY OF GROSSI AND VITTER

This structure [Grossi and Vitter 2000; 2005] builds a hierarchy of samplings over A. At the first level 0, it treats differently the even and odd values of $A_0 = A$. If $A_0[i]$ is even, then it stores its value divided by 2 in the next suffix array A_1. If $A_0[i]$



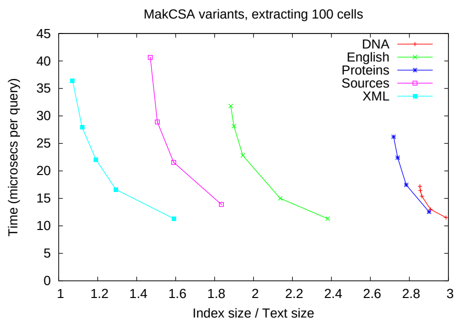


Fig. 9. Performance of *MakCSA* over the different texts, using in each case the C value that optimizes the extraction of one cell. On the left, extraction of one cell, on the right, of 100 cells.

is odd, then it stores in an array Ψ_0 the position where $A_0[i]+1$ occurs in A_0. The lowest bits of $A_0[1,n]$ (i.e., those that determine whether $A_0[i]$ is even or odd) are concatenated into a bitmap $B_0[1,n]$. Therefore, if $B_0[i]=0$, we know that $A[i] = A_0[i] = 2\cdot A_1[rank_0(B_0,i)]$. Else, we have $A[i]=A_0[i]=A_0[\Psi_0[rank_1(B_0,i)]]-1$, and this second A_0 cell must be even.

This second array, $A_1[1,n_1]$, $n_1 = n/2$, is recursively represented in the same way, and this continues up to level t, where $A_t[1,n_t]$ is explicitly represented.

The main issue is how to represent the arrays $\Psi_k[1,n_k/2]$, where $n_k = n/2^k$. They show that any Ψ_k can be seen as the concatenation of σ^{2^k} lists of increasing values. As these values are in the range $[1,n_k]$, they make the whole list increasing by adding $j\cdot n_k$ to the elements of the j-th list. The result is an increasing list of $n_k/2$ increasing numbers in the range $[1,\sigma^{2^k} n/2^k]$.

They resort to a bitmap representation for increasing sequences [Fano 1971; Elias 1974; Okanohara and Sadakane 2007]. To represent m increasing numbers $X[1,m]$ on $[1,n]$, their $b=\lceil \lg(n/m)\rceil$ lowest bits are represented explicitly in an array $L[1,m]$, whereas the higher bits are represented differentially and using unary encoding in a bitmap $H[1,2n]$. That is, let h_i and h_{i-1} be the $\lceil \lg n\rceil - b$ highest bits of $X[i]$ and $X[i-1]$, respectively, then we append $1^{h_i-h_{i-1}}0$ to H. Then it holds $X[i] = L[i] + 2^b(select_0(H,i)-i)$.

As a result, Ψ_k is represented using $(n_k/2)\lg((\sigma^{2^k} n_k)/(n_k/2))+O(n_k) = (n/2)\lg\sigma$ $+O(n/2^k)$ bits. Added over t levels, we have space $t(n/2)\lg\sigma + O(n) + n_t \lg n_t$, the latter term being for the explicit representation of A_t. One can choose $t=\lg\lg_\sigma n$ to obtain $(n/2)\lg\sigma\lg\lg_\sigma n + O(n\lg\sigma)$ bits of space and $O(t)$ time to access $A[i]$.

The idea can be generalized as follows. Choose the values of A_k that are multiple of a parameter ℓ, to be copied (divided by ℓ) to level $k+1$. Mark $B_k[i]=0$ iff $A_k[i]$ is a multiple of ℓ. Store in $\Psi_k[1,(1-1/\ell)n_k]$ all the values where $B_k[i]=1$. Now we have to access Ψ_k up to $\ell-1$ times before proceeding to the next level. The analysis is very similar, except that 2^k is replaced by ℓ^k. Thus the total space is $(1-1/\ell)n\lg\sigma\lg_\ell\lg_\sigma n + O(n\lg\sigma)$ bits, with $t=\lg_\ell\lg_\sigma n$. The time is $O(\ell t)$. In particular, choosing $\ell = \lg_\sigma^\epsilon n$ for a constant $0<\epsilon<1$, we have $t=1/\epsilon$, the space is $(1/\epsilon + O(1))n\lg\sigma$ bits and the time is $O((1/\epsilon)\lg^\epsilon n)$.

We implemented a verbatim variant of this data structure, where we used sparse bitmap implementations from the *libcds* library.[11] Note that one can access any position of any list of Ψ_k in constant time by knowing the list number and offset. However, to know the list number we need to know the positions in A where the suffix beginning with any tuple of Σ^k starts. This requires $\sigma^k \lg n$ additional bits.

We implemented a second variant of this structure. Instead of using the Σ^k contexts (many of which may actually be empty), we detect maximal runs of increasing numbers in Ψ_k and take those as the lists. The beginning of the lists are marked in a sparse bitmap $S_k[1, n_k]$. Then, in order to retrieve $\Psi_k[i]$ we compute $j = rank_1(S_k, i)$, to find that i belongs to the j-th list, and use the same numbering scheme as before: since $\Psi_k[i]$ is represented as $X[i] = j \cdot n_k + \Psi_k[i]$, we compute $X[i]$ from the representation using H and L, and then subtract $j \cdot n_k$. The space for S_k is just $O(n_k)$ bits, and thus the space and time analysis stays the same.

Figure 10 compares various (t, ℓ) combinations for both variants of the structure (the basic one and the one using runs), on a couple of texts; the results are similar for the rest. It is clear that the best tradeoffs are obtained when using $t = 1$, that is, not using a recursive structure but just one level of sampling, and then storing the samples in plain form. Space can be reduced by using a larger t (i.e., more levels of recursions), but it is always faster to reduce the same space by using a larger ℓ value (i.e., a sparser sampling at the first level). Only on `dna` there are some dominating points using $t = 2$. It is also clear that our variant using runs is much better when using $t > 1$ levels (indeed, the basic variant is almost never affordable for $t > 2$), but there is almost no difference between variants on $t = 1$.

Figure 11 (left) compares the two variants, for all the texts, using the dominating points of each scheme (mostly corresponding to $t = 1$, as mentioned). It can be seen that, when using only one level, the differences are minimal. We will use the version with runs as the representative of *GVCSA* in the main experiments.

D. IMPLEMENTING THE SUFFIX ARRAY OF RAO

The compressed suffix array of Rao [2002] looks in principle similar to the generalized structure of Grossi and Vitter, yet there are key subtle differences. The main idea is that, instead of iterating up to $\ell - 1$ times in a level k before moving to level $k+1$, he stores vectors $d_k[1, n_k]$ with the value $d_k[i] = (\ell - 1) - ((A_k[i] - 1) \bmod \ell)$, that is, at which distance is the value $A_k[i]$ from the next multiple of ℓ. The idea is then that $\Psi_k[i]$ stores the position in A_k where the value $A_k[i] + d_k[i]$ is stored, and therefore we jump directly to the cell with an answer in one step, not $d_k[i]$ steps. The final answer is thus $A_k[\Psi_k[i]] - d_k[i]$.

Unfortunately, this new Ψ_k array does not enjoy the monotonicity properties seen before; these hold only within the subsequence associated to a single value of d_k. Thus the array is split into $\ell - 1$ arrays Ψ_k^δ, $1 \le \delta < \ell$, containing the values $\Psi_k[i]$ such that $d_k[i] = \delta$. At level k, we store $\ell - 1$ bitmaps $V_k^\delta[1, n_k]$ where $V_k^\delta[i] = 1$ iff $d_k[i] = \delta$. Then the value $\Psi_k[i]$ is found at $\Psi_k^\delta[rank_1(V_k^\delta, i)]$ if $d_k[i] = \delta$.

Note that Ψ_k^δ is formed by $\sigma^{\delta \ell^k}$ increasing lists on $[1, n_k]$. The renumbering scheme of the previous section[12] yields a single list of n_k/ℓ values in $[1, \sigma^{\delta \ell^k} n_k]$, with

[11] `https://github.com/fclaude/libcds`

[12] In fact they use a slightly different mechanism, but the complexity does not change.

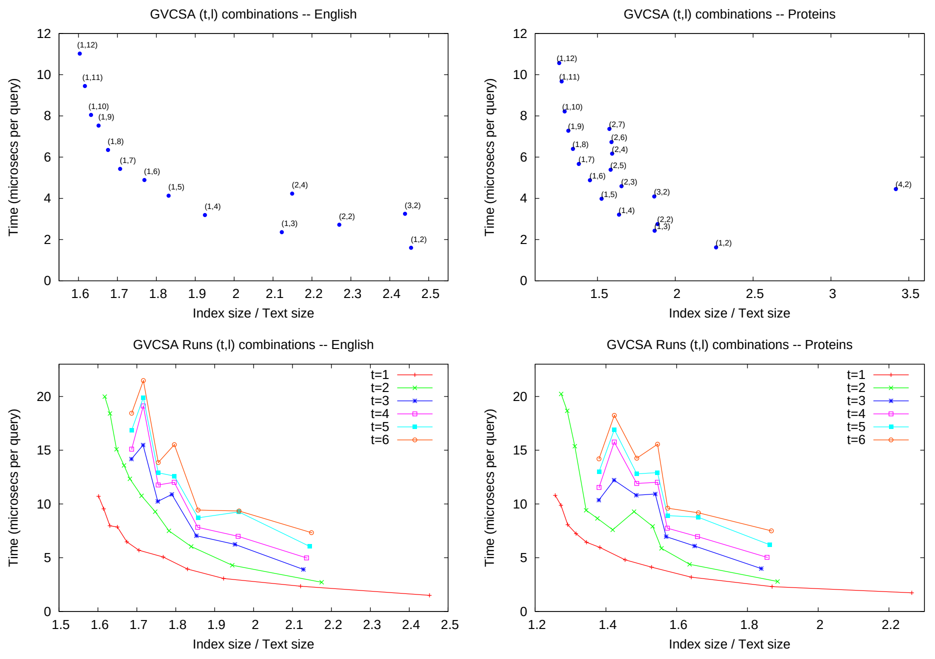


Fig. 10. Space/time tradeoffs for accessing one cell using various options for (t, ℓ) for *GVCSA*. On top the basic scheme and on the bottom our improvement using runs. On the left, on `english` text; on the right, on `proteins`. On the bottom we show one curve per t value; the results with ℓ value from 2 onwards are shown right to left in the curve.

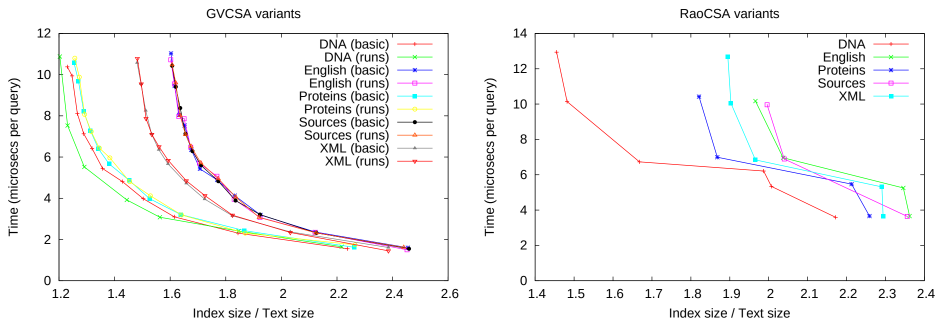


Fig. 11. Time-space tradeoffs to access one cell. On the left, basic *GVCSA* versus the version with runs, for all the texts. On the right, the best variants of *RaoCSA*.

$n_k = n/\ell^k$. The representation using L and H then yields $(\delta/\ell)n \lg \sigma + O((n_k/\ell)\lg \ell)$ bits. Summing for $1 \le \delta < \ell$, at level k we have $O(\ell n \lg \sigma + n_k \lg \ell)$ bits for the lists. Vector d adds $O(n_k \lg \ell)$ bits and vectors V add $O(\ell n_k)$ bits, for a total of $O(\ell n \lg \sigma + \ell n_k)$ bits at level k. Added over t levels and considering the final explicit array A_t we have $O(t \ell n \lg \sigma) + O(\ell n) + O((n/\ell^t)\lg n)$ bits. Choosing $\ell = \lg_\sigma^{1/t} n$ we get space $O(t \lg_\sigma^{1/t} n)\ n \lg \sigma$ bits, and $O(t)$ time to compute any $A[i]$. This gives a number of space/time tradeoffs, consider for example $t = 1/\epsilon$ or $t = \lg\lg_\sigma n$.

In practice the V_k^δ bitmaps may occupy too much space. We implement a second variant where we completely remove them, and instead represent vector d as a wavelet tree [Grossi et al. 2003], implemented in library *libcds*. This occupies $O(n_k \lg \ell)$ bits of space instead of $O(\ell n_k)$, and supports $rank_\delta(d_k, i) = rank_1(V_k^\delta, i)$ within $O(\lg \ell)$ time instead of $O(1)$. In theory, the asymptotic space does not change and the access time grows to $O(t \lg \ell) = O(\lg\lg_\sigma n)$, which is a mild growth. In practice, this is advantageous, as we see soon.

We also consider a variant replacing the strict numbering by runs, as for *GVCSA*.

Figure 12 shows the space/time tradeoffs obtained to access a random cell using various (t, ℓ) combinations for this index. We show the results on `english` and `proteins`; the results are similar for the rest. On top we show the basic scheme, where it always holds that the combination $(t = 1, \ell = 2)$ dominates all the others (note this combination corresponds to *GVCSA*). On the middle we show the scheme where the bitmaps are replaced with wavelet trees. In this case the combination $(t = 2, \ell = 2)$ offers better space sometimes. On the bottom we show the improvement including runs and wavelet trees. This time many more (t, ℓ) combinations are feasible, and various alternatives with $\ell = 2$ (and even $\ell = 3$) offer relevant space/time tradeoffs.

It is also clear that the variant with wavelet trees and runs is always the best. Figure 11 (right) shows the results obtained choosing the dominating (t, ℓ) combinations of this variant, for all the texts. Those will be used to represent *RaoCSA* in the main experiments.

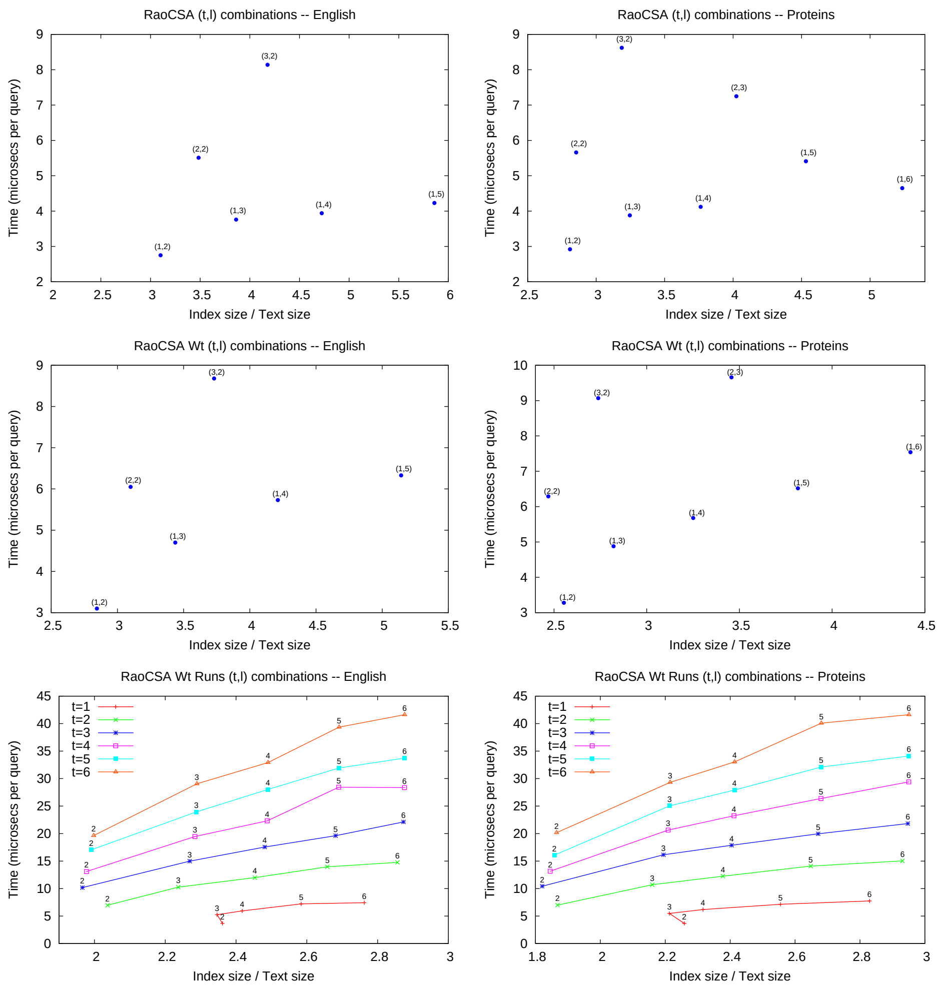


Fig. 12. Various options for (t, ℓ) for RaoCSA. On top the basic scheme, on the middle our improvement using wavelet trees, and on the bottom our improvement using runs and wavelet trees. On the left, on `english` text; on the right, on proteins. On the bottom we show one curve per t value; the ℓ values are marked in the curves.