Algorithms **2013**, *6*, 319-351; doi:10.3390/a6020319

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Practical Compressed Suffix Trees

Andrés Abeliuk [1], Rodrigo Cánovas [2] and Gonzalo Navarro [1,*]

[1] Department of Computer Science, University of Chile, Santiago 8320000, Chile; E-Mail: aabeliuk@dcc.uchile.cl

[2] NICTA Victoria Research Laboratory, Department of Computing and Information Systems; University of Melbourne, Victoria 3010, Australia; E-Mail: rcanovas@student.unimelb.edu.au

* Author to whom correspondence should be addressed; E-Mail: gnavarro@dcc.uchile.cl; Tel.: +56-2-29784952; Fax: +56-2-26895531.

Received: 18 March 2013; in revised form: 24 April 2013 / Accepted: 26 April 2013 / Published: 21 May 2013

Abstract: The suffix tree is an extremely important data structure in bioinformatics. Classical implementations require much space, which renders them useless to handle large sequence collections. Recent research has obtained various compressed representations for suffix trees, with widely different space-time tradeoffs. In this paper we show how the use of *range min-max trees* yields novel representations achieving practical space/time tradeoffs. In addition, we show how those trees can be modified to index highly repetitive collections, obtaining the first compressed suffix tree representation that effectively adapts to that scenario.

Keywords: suffix trees; compressed data structures; repetitive sequence collections; bioinformatics

---

1. Introduction

As a result of the sharply falling costs of sequencing, large sequence databases are rapidly emerging, and they will soon grow to thousands of genomes and more. Sequencing machines are already producing terabytes of data per day, at a few thousand dollars per genome, and will cost less very soon (see, e.g., [1]). This raises a number of challenges in bioinformatics, particularly because those sequence databases are meant to be the subject of various analysis processes that require sophisticated data structures built on them.

The *suffix tree* [2–4] is probably the most important of those data structures. Many complex sequence analysis problems are solved through sophisticated traversals over the suffix tree [5]. However, a serious problem of suffix trees, aggravated on large sequence collections, is that they take much space. A naive implementation can easily require 20 bytes per character, and a very optimized one reaches 10 bytes [6]. A way to reduce this space to about 4 bytes per character is to use a simplified structure called a *suffix array* [7]. However, suffix arrays do not contain sufficient information to carry out all the complex tasks suffix trees are used for. Enhanced suffix arrays [8] extend suffix arrays so as to recover the full suffix tree functionality, raising the space to about 6 bytes per character in practice. Other heuristic space-saving methods [9] achieve about the same.

For example, on DNA sequences, each character could be encoded with just 2 bits, whereas the alternatives we have considered require 32 to 160 bits per character (bpc). This is an overhead of 1600% to 8000%! The situation is also a heresy in terms of Information Theory: Whereas the information contained in a sequence of n symbols over an alphabet of size σ is $n \log \sigma$ bits in the worst case, all the alternatives above require $\Theta(n \log n)$ bits. (Our logarithms are in base 2)

A solution to the large space requirements is of course resorting to secondary memory. However, using suffix trees on secondary memory makes them orders of magnitude slower, as most of the required traversals are highly non-local. A more interesting research direction, which has made much progress in recent years, is to design *compressed suffix trees* (*CSTs*). Those compressed representations of suffix trees are able to approach not only the worst-case space of the sequence, but even its information content (*i.e.*, they approach the space of the *compressed* sequence). CSTs are composed of a *compressed suffix array* (*CSA*) plus some extra information that encodes the suffix tree topology and *longest common prefix* (*LCP*) information (more details later). The existing solutions can be divided into three categories:

Explicit topology. The first CST proposal was by Sadakane [10,11]. It uses $4n+o(n)$ bits to represent the suffix tree topology, plus $2n + o(n)$ bits to represent the LCP values. In addition, it uses Sadakane's CSA [12], which requires $nH_0 + O(n \log \log \sigma)$ bits, where H_0 is the zero-order entropy of the sequence. This structure supports most of the tree navigation operations in constant time (except, notably, going down to a child, which is an important operation), at the price of $\Theta(n)$ bits of space on top of a CSA.

Sampling. A second proposal was by Russo *et al.* [13,14]. It is based on sampling some suffix tree nodes and recovering the information on any other node by moving to the nearest sample using suffix links. It requires only $o(n)$ bits on top of a CSA. By using an FM-index [15] as the CSA, one achieves $nH_k + o(n \log \sigma)$ bits of space, where H_k is the k-th order empirical entropy of the sequence (a measure of statistical compressibility [16]), for sufficiently low $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. The navigation operations are supported in polylogarithmic time (at best $\Theta(\log n \log \log n)$ in their paper).

Intervals. Yet a third proposal was by Fischer *et al.* [17–19]. It avoids the explicit representation of the tree topology by working all the time with the corresponding suffix array intervals. All the suffix tree navigation is reduced to three primitive operations on such intervals: Next/Previous smaller value and range minimum queries (more details later). The proposal achieves a space/time tradeoff that is between the two previous ones: It reduces the $\Theta(n)$ extra bits of Sadakane to $o(n)$, and the superlogarithmic operation times of Russo *et al.* to $O(\log^\epsilon n)$, for any constant $0 < \epsilon < 1$. More precisely, the space is $nH_k(2 \max(1, \log(1/H_k)) + 1/\epsilon + O(1)) + o(n \log \sigma)$ bits (for the same k ranges as above).

We remark that the space figures include the storage of the sequence itself, as all modern CSAs are *self-indexes*, that is, they contain both the sequence and its suffix array.

For applications, the *practical* performance of the above schemes is more relevant than the theoretical figures. The solution based on explicit topology was implemented by Välimäki *et al*. [20]. As expected from theory, the structure is very fast, achieving a few tens of microseconds per operation, but uses significant space (about 25–35 bpc, close to a suffix array). This undermines its applicability. Very recently, Gog [21] showed that this implementation can be made much more space-efficient, using about 13–17 bpc. This still puts this solution in the range of the "large" CSTs in the context of this paper.

The structure based on sampling was implemented by Russo. As expected from theory again, it was shown to achieve very little space, around 4–6 bpc, which makes it extremely attractive when the sequence is large compared with the available main memory. On the other hand, the structure is much slower than the previous one: Each navigation operation takes the order of milliseconds.

In this paper we present the *first* implementations of the third approach, based on intervals. As predicted by theory once again, we achieve practical implementations that lie between the two previous extremes (too large or too slow) and offer attractive space/time tradeoffs. One variant shows to be superior to the original implementation of Sadakane's CST [20] in both space and time: It uses 13–16 bpc (*i.e*., half the space) and requires a few microseconds per operation (*i.e*., several times faster). A second variant works within 8–12 bpc and requires a few hundreds of microseconds per operation, that is, smaller than our first variant and still several times faster than Russo's implementation. We remark again that those space figures include the storage of the sequence, and that 8 bpc is the storage space of a byte-based representation of the plain sequence (although 2 bpc is easily achieved on DNA).

Achieving those good practical results is not immediate, however. We show that a direct implementation of the techniques as described in the theoretical proposal [18] does not lead to the best performance. Instead, we propose a novel solution to the previous/next smaller value and range minimum queries, based on the *range min-max tree* (*RMM tree*), a recent data structure developed for succinct tree representations [22,23]. We adapt RMM trees to speed up the desired queries on sequences of LCP values.

The resulting representation not only performs better than a verbatim implementation of the theoretical proposal, but also has a great potential to handle *highly repetitive* sequence collections. Those collections, formed by many similar strings, arise in version control systems, periodic publications, and software repositories. They also arise naturally in bioinformatics, for example when sequencing the genomes of many individuals of the same or related species (two human genomes share 99.9% of their sequences, for example). It is likely that the largest genome collections that will emerge in the next years will be highly repetitive, and taking advantage of that repetitiveness will be the key to handle them.

Repetitiveness is not captured by statistical compression methods nor frequency-based entropy definitions [16,24] (*i.e*., the frequencies of symbols do not change much if we add near-copies of an initial sequence). Rather, we need *repetition aware* compression methods. Although this kind of compression is well-known (e.g., grammar-based and Ziv-Lempel-based compression), only recently there have appeared CSAs and other indexes that take advantage of repetitiveness [24–27]. Yet, those indexes do not support the full suffix tree functionality. On the other hand, none of the existing CSTs is tailored to repetitive text collections.

Our second contribution is to present the *first*, and the only to date, fully-functional compressed suffix tree whose compression effectiveness is related to the repetitiveness of the text collection. While its operations are much slower than most existing CSTs (requiring the order of milliseconds), the space required is also much lower on repetitive collections (1.4–1.9 bpc in our mildly repetitive test collections, and 0.6 bpc on a highly repetitive one). This can make the difference between fitting the collection in main memory or having to resort to disk.

The paper is organized as follows. Section 2 gives the basis of suffix array and tree compression in general and repetitive text collections, and puts our contribution in context. Section 3 compares various LCP representations and chooses suitable ones for our CST. Section 4 proposes the novel RMM-tree based solution for *NSV*/*PSV*/*RMQ* operations and shows it is superior to a verbatim implementation of the theoretical proposal. Section 5 assembles our CST using the previous pieces, and Section 6 evaluates and compares it with alternative implementations. Section 7 extends the RMM concept to design a CST that adapts to repetitive collections, and Section 8 evaluates it. Finally, Section 9 concludes and discusses various new advances [21,28] that have been made on efficiently implementing the interval-based approach since the conference publication of our results [29,30].

2. Related Work and Our Contribution in Context

Our aim is to handle a *collection* of texts $T_1, T_2, \ldots$ over an alphabet Σ of size σ. This is customarily represented as a single text $T = T_1\$T_2\$\ldots$ concatenating all the texts in the collection. The symbol "$" is an endmarker that does not appear elsewhere. Such a representation simplifies the discussion on data structures, which can consider that a single text is handled.

2.1. Suffix Arrays and CSAs

A *suffix array* over a text $T[1,n]$ is an array $A[1,n]$ of the positions in T, lexicographically sorted by the suffix starting at the corresponding position of T. That is, $T[A[i],n] < T[A[i+1],n]$ for all $1 \leq i < n$. Note that every substring of T is the prefix of a suffix, and that all the suffixes starting with a given pattern P appear consecutively in A, hence a pair of binary searches find the area $A[sp,ep]$ containing all the positions where P occurs in T.

There are several *compressed suffix arrays* (*CSAs*) [31,32], which offer essentially the following functionality: (1) Given a pattern $P[1,m]$, find the interval $A[sp,ep]$ of the suffixes starting with P; (2) Obtain $A[i]$ given i; (3) Obtain $A^{-1}[j]$ given j. An important function the CSAs implement is $\Psi(i) = A^{-1}[(A[i] \bmod n) + 1]$ and its inverse, usually much faster than computing A and A^{-1}. This function lets us move virtually in the text, from the suffix i that points to text position $j = A[i]$, to the one pointing to $j+1 = A[\Psi(i)]$. Function Ψ is essential to support suffix tree functionality.

Most CSAs are *self-indexes*, meaning that they can also extract any substring of T and thus T does not need to be stored separately. Generally, one can extract any $T[i,i+\ell]$ using $s+\ell$ applications of Ψ, where s is a sampling parameter that involves spending $O((n/s)\log n)$ extra bits of space.

2.2. Suffix Trees and CSTs

A *suffix trie* is a trie (or digital tree) storing all the suffixes of T. This is a labeled tree where each text suffix is read in a root-to-leaf path, and the edges toward the children of a node are labeled by different characters. The concatenation of the string labels from the root up to a node v is called the *path-label* of v, $\pi(v)$. A *suffix tree* is obtained by "compacting" unary paths in a suffix trie, which means they are converted into a single edge labeled by the string that concatenates the involved character labels. Furthermore, a node v is converted into a leaf (and the rest of its descending unary path is eliminated) as soon as $\pi(v)$ becomes unique. Such leaf is annotated with the starting position i of the only occurrence of $\pi(v)$ in T, that is, suffix $T[i, n]$ starts with $\pi(v)$. If the children of each suffix tree node are ordered lexicographically by their string label, then the sequence of leaf annotations of the suffix tree is precisely the suffix array of T. Figure 1 illustrates a suffix tree and suffix array. Several navigation operations over the nodes and leaves of the suffix tree are of interest. Table 1 lists the most common ones.

Figure 1. The suffix tree of the text "alabar a la alabarda$", where the "$" is a terminator symbol. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters "a"–"z".

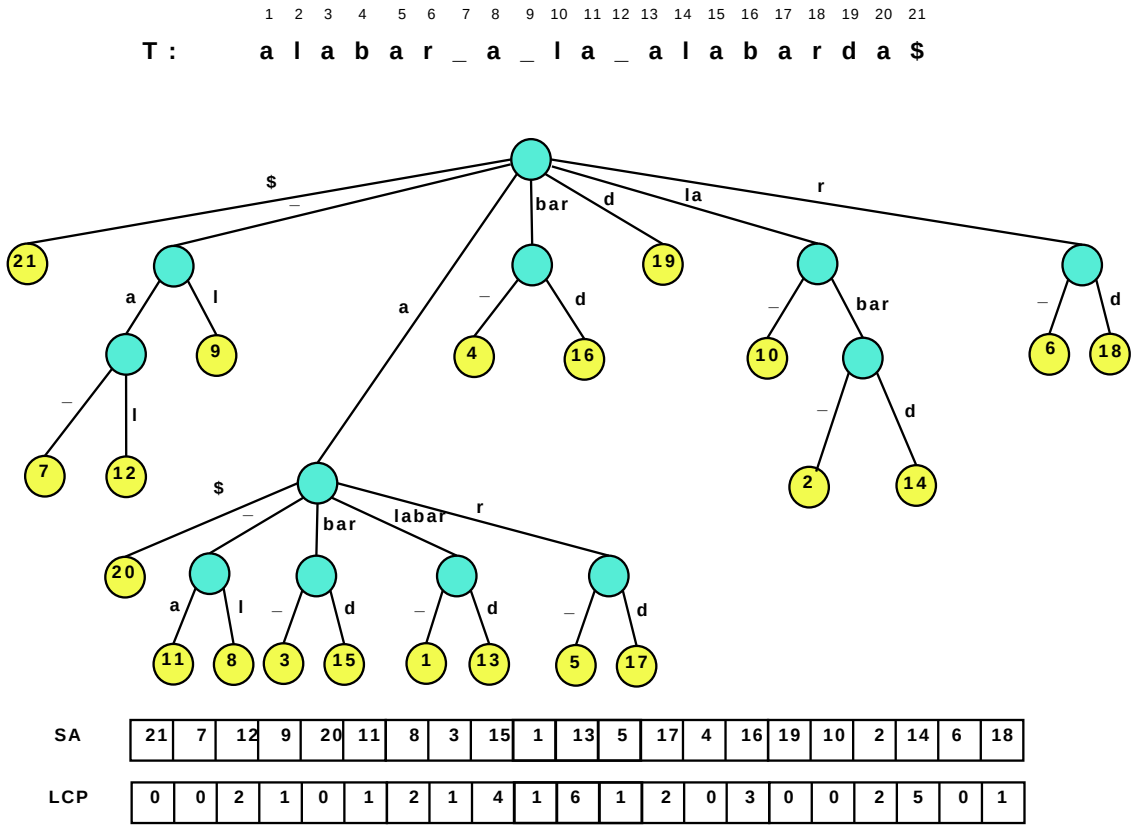


Table 1. Main operations over the nodes and leaves of the suffix tree.

Operation	**Description**
Root()	the root of the suffix tree
Locate(v)	position i such that $\pi(v)$ starts at $T[i]$, for a leaf v
Ancestor(v, w)	true if v is an ancestor of w
SDepth(v)	string-depth of v, *i.e.*, $\|\pi(v)\|$
TDepth(v)	tree-depth of v, *i.e.*, depth of node v in the suffix tree
Count(v)	number of leaves in the subtree rooted at v
Parent(v)	parent node of v
FChild(v)	alphabetically first child of v
NSibling(v)	alphabetically next sibling of v
SLink(v)	suffix-link of v, *i.e.*, the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma$
SLink^i(v)	iterated suffix-link, *i.e.*, the node w s.th. $\pi(w) = \beta$ if $\pi(v) = \alpha\beta$ for $\alpha \in \Sigma^i$
LCA(v, w)	lowest common ancestor of v and w
Child(v, a)	node w s.th. the first letter on edge (v, w) is $a \in \Sigma$
Letter(v, i)	ith letter of v's path-label, $\pi(v)[i]$
LAQ$_S$(v, d)	the highest ancestor of v with string-depth $\geq d$
LAQ$_T$(v, d)	the ancestor of v with tree-depth d

The suffix array alone is insufficient to efficiently emulate all the relevant suffix tree operations. To recover the full suffix tree functionality, we need two extra pieces of information: (1) The tree topology; (2) The *longest common prefix* (*LCP*) information, that is, *LCP*$[i]$ is the length of the longest common prefix between $T[A[i-1], n]$ and $T[A[i], n]$ for $i > 1$ and *LCP*$[1] = 0$ (or, seen another way, *LCP*$[i]$ is the length of the string labeling the path from the root to the lowest common ancestor node of the ith and $(i-1)$th suffix tree leaves). Indeed, the suffix tree topology can be implicit if we identify each suffix tree node with the suffix array interval containing the leaves that descend from it. This range uniquely identifies the node because there are no unary nodes in a suffix tree.

Consequently, a *compressed suffix tree* (*CST*) is obtained by enriching the CSA with some extra data. Sadakane [11] added the topology of the tree (using $4n$ extra bits) and the LCP data. The LCP data was compressed to $2n$ bits by noticing that, if sorted by text order rather than suffix array order, the LCP numbers decrease by at most 1: Let *PLCP* be the permuted *LCP* array, then *PLCP*$[j+1] \geq$ *PLCP*$[j] - 1$. Thus the numbers can be differentially encoded, $h[j+1] =$ *PLCP*$[j+1] -$ *PLCP*$[j] + 1 \geq 0$, and then represented in unary over a bitmap $H[1, 2n] = 0^{h[1]}10^{h[2]} \ldots 10^{h[n]}1$. Then, to obtain *LCP*$[i]$, we look for *PLCP*$[A[i]]$, and this is extracted from H via *rank*/*select* operations. Here $rank_b(H, i)$ counts the number of bits b in $H[1, i]$ and $select_b(H, i)$ is the position of the i-th b in H. Both can be answered in constant time using $o(n)$ extra bits of space [33]. Then *PLCP*$[j] = select_1(H, j) - 2j$, assuming *PLCP*$[0] = 0$.

Russo *et al.* [14] get rid of the parentheses, by instead identifying suffix tree nodes with their corresponding suffix array interval. By sampling some suffix tree nodes, most operations can be carried out by moving, using suffix links, towards a sampled node, finding the information stored in there, and

transforming it as we move back to the original node. The suffix link operation, defined in Table 1, can be computed using Ψ and the lowest common ancestor operation [11].

2.3. Re-Pair and Repetition-Aware CSAs

Re-Pair [34] is a grammar-based compression method that factors out repetitions in a sequence. It is based on the following heuristic: (1) Find the most repeated pair ab in the sequence; (2) Replace all its occurrences by a new symbol s; (3) Add a rule $s \rightarrow ab$ to a dictionary R; (4) Iterate until every pair is unique.

The result of the compression of a text T, over an alphabet Σ of size σ, is the dictionary R and the remaining sequence C, containing new symbols (s) and symbols in Σ. Every sub-sequence of C can be decompressed locally by the following procedure: Check if $C[i] < \sigma$; if so the symbol is original, else look in R for rule $C[i] \rightarrow ab$, and recursively continue expanding with the same steps.

The dictionary R corresponds to a context-free grammar, and the sequence C to the initial symbols of the derivation tree that represents T. The final structure can be regarded as a sequence of binary trees with roots $C[i]$.

González and Navarro [35] used Re-Pair to compress the differentially encoded suffix array, $A'[i] = A[i] - A[i-1]$. They showed that Re-Pair achieves $|R| + |C| = O(r \log \frac{n}{r})$ on A', r being the number of *runs in* Ψ. A run in Ψ is a maximal contiguous area where $\Psi(i+1) = \Psi(i) + 1$. It was shown that the number of runs in Ψ is $r \leq nH_k + \sigma^k$ for any k [36]. More importantly, repetitions in T induce long runs in Ψ, and hence a smaller r [25]. An exact bound has been elusive, but Mäkinen *et al.* [25] gave an average-case upper bound for r: If T is formed by a random base sequence of length $n' \ll n$ and then other sequences that have m random *mutations* (which include indels, replacements, block moves, *etc.*) with respect to the base sequence, then r is at most $n' + O(m \log_\sigma n)$ on average.

The RLCSA [25] is a CSA where those runs in Ψ are factored out, to achieve $O(r)$ cells of space. More precisely, the size of the RLCSA is $r(2\log(n/r) + \log\sigma)(1 + o(1))$ bits. It supports accesses to A in time $O(s \log n)$, with $O((n/s)\log n)$ extra bits for a sampling of A.

2.4. An Interval-Based CST

Fischer *et al.* [18] prove that bitmap H in Sadakane's CST is compressible as it has at most $2r$ *runs* of 0's or 1's, where r is the number of runs in Ψ as before.

Let $z_1, z_2, \ldots, z_r$ be the lengths of the runs of 0's and $o_1, o_2, \ldots, o_r$ be the lengths of the runs of 1's. Fischer *et al.* [18] create arrays $Z = 10^{z_1-1}10^{z_2-1}\ldots$ and $O = 10^{o_1-1}10^{o_2-1}\ldots$, with overall $2r$ 1's out of $2n$, and thus can be compressed to $2r \log \frac{n}{r} + O(r) + o(n)$ bits while supporting constant-time *rank* and *select* [37]. When r is very small it is better to use other bitmap representations (e.g., [38]) requiring $2r \log \frac{n}{r} + O(r)$ bits, even if they do not offer constant times for *rank*.

Their other improvement over Sadakane's CST is to get rid of the tree topology and replace it with suffix array ranges. Fischer *et al.* show that all the navigation can be simulated by means of three operations: (1) The range minimum query *RMQ*(i, j) gives the position of the minimum in *LCP*$[i, j]$; (2) The previous smaller value *PSV*(i) finds the last value smaller than *LCP*$[i]$ in *LCP*$[1, i-1]$; and

(3) The next smaller value $NSV(i)$ finds the first value smaller than $LCP[i]$ in $LCP[i+1, n]$. As examples, the parent of node $[i, j]$ can be computed as $[PSV(i), NSV(i) - 1]$; the LCA between nodes $[i, j]$ and $[i', j']$ is $[PSV(p), NSV(p) - 1]$, where $p = RMQ(\min(i, i'), \max(j, j'))$; and the suffix link of $[i, j]$ is $[PSV(\Psi(i)), NSV(\Psi(j)) - 1]$.

The three primitives could easily be solved in constant time using $O(n)$ extra bits of space on top of the *LCP* representation [18] (more recently Ohlebusch *et al.* [28] showed how to solve them in constant time using $3n + o(n)$ bits), but Fischer *et al.* give sublogarithmic-time algorithms to solve them with only $o(n)$ extra bits.

A final observation of interest in that article [18] is that the repetitions found in a differential encoding of the suffix array (*i.e.*, the runs in Ψ) also show up in the differential *LCP*, since if A values differ by 1 in a range, the *LCP* areas must also differ by 1. Therefore, Re-Pair compression of the differential *LCP* should perform similarly as González and Navarro's suffix array compression [35].

2.5. Our Contribution

In this paper we design a practical and efficient CST building on the ideas of Fischer *et al.* [18], and show that its space/time performance is attractive. This challenge can be divided into (1) How to represent the LCP array efficiently in practice; and (2) How to compute efficiently *RMQ*, *PSV*, and *NSV* over this LCP representation. For the first sub-problem we compare various implementations, whereas for the second we adapt a data structure called range min-max tree [22], which was designed for another problem. We compare the resulting CST with previous ones and show that it offers relevant space/time tradeoffs. Finally, we design a variant of the range min-max tree that is suitable for highly repetitive collections (*i.e.*, with low r value). This is obtained by replacing this regular tree by the (truncated) grammar tree of the Re-Pair compression of the differential LCP array. The resulting CST is the only one handling repetitive collections within space below 2 bpc.

3. Representing Array *LCP*

The following alternatives were considered to represent *LCP*:

Sad-Gon Encodes Sadakane's [11] H bitmap in plain form, using the *rank*/*select* implementation of González [39], which takes $0.1n$ bits on top of the $2n$ used by H itself. This implementation answers *rank* in constant time and *select* in $O(\log n)$ time via binary search.

Sad-OS Like the previous one, but using the *dense array* implementation of Okanohara and Sadakane [38] for H. This requires about the same space as the previous one and answers *select* in time $O(\log^4 r / \log n)$.

FMN-RRR Encodes H in compressed form as in Fischer *et al.* [18], that is, by encoding sparse bitmaps Z and O. We use the compressed representation by Raman *et al.* [37] as implemented by Claude [40]. This costs $0.54n$ extra bits on top of the entropy of the two bitmaps, $2r \log \frac{n}{r} + O(r)$. Operation *rank* takes constant time and *select* takes $O(\log n)$ time.

FMN-OS Like the previous one, but instead of Raman *et al.*'s technique, we use the *sparse array* implementation by Okanohara and Sadakane [38]. This requires $2r \log \frac{n}{r} + O(r)$ bits and solves *select* in time $O(\log^4 r / \log n)$.

PT Inspired in an LCP construction algorithm [41], we store a particular sampling of *LCP* values, and compute the others using the sampled ones. Given a parameter v, the sampling requires $n+O(n/\sqrt{v}+v)$ bytes of space and computes any $LCP[i]$ by comparing at most some $T[j, j+v]$ and $T[j', j'+v]$. As we must obtain these symbols using Ψ up to $2v$ times, the idea is slow.

PhiSpare This is inspired in another construction [42]. For a parameter q, we store in text order an array $PLCP_q$ with the *LCP* values for all text positions $q \cdot k$. Now assume $A[i] = qk + b$, with $0 \leq b < k$. If $b = 0$, then $LCP[i] = PLCP_q[k]$. Otherwise, $LCP[i]$ is computed by comparing at most $q + PLCP_q[k+1] - PLCP_q[k]$ symbols of the suffixes $T[A[i-1], n]$ and $T[A[i], n]$. The space is n/q integers and the computation requires $O(q)$ applications of Ψ on average.

DAC The *directly addressable codes* of Ladra *et al.* [43]. Most *LCP* values are small ($O(\log_\sigma n)$ on average), and thus require few bits. Yet, some can be much larger. Thus we can fix a block length b and divide each number of ℓ bits into $\lceil \ell/b \rceil$ blocks of b bits. Each block is stored using $b+1$ bits, the last one telling whether the number continues in the next block or finishes in the current one. Those blocks are then rearranged to allow for fast random access. There are two variants of this structure, both implemented by Ladra: One with fixed b (*DAC*), and another using different b values for the first, second, *etc.* blocks, so as to minimize the total space (*DAC-Var*). Note we represent *LCP* and not *PLCP*, thus we do not need to compute $A[i]$.

RP *Re-Pair* As mentioned, Re-Pair can be used to compress the differential *LCP* array, *LCP*′ [18]. To obtain $LCP[i]$ we store sampled absolute *LCP* values and decompress the nonterminals since the last sample.

3.1. Experimental Comparison

Our computer is an Intel Core2 Duo at 3.16 GHz, with 8 GB of RAM and 6 MB cache, running Linux version 2.6.24-24. Table 2 lists the collections used for this experiment. They were obtained from the *Pizza & Chili* site [44]. We also give some information on their LCP values and number of runs in Ψ, which affects their compressibility. We tested the different LCP representations by accessing 100,000 random positions of the *LCP* array. Figure 2 (left) shows the space/times achieved on the four texts. Only PT and *PhiSpare* display a space/time tradeoff; in the first we use $v = 4, 6, 8$ and for the second $q = 16, 32, 64$. Because in various cases we need to access a sequence of consecutive LCP values, we show in Figure 2 (right) the time per cell when accessing 32 consecutive cells. The plots also show solution *Naive*, which stores LCP values using the number of bits required by the maximum value in the array.

Table 2. Data files used for tests with their size in MB, plus some information on their compressibility.

Collection	Size	Description	Avg LCP	Max LCP	Runs (r/n)
`dna`	100	DNA sequences from Gutenberg Project	28.1	17,772	0.63
`xml`	100	XML from dblp.uni-trier.de	44.5	1084	0.14
`proteins`	100	Protein sequences from the Swissprot database	220.6	35,246	0.59
`sources`	100	Source code obtained by concatenating files of the Linux-2.6.11.6 and gcc-4.0.0 distributions	625.1	307,871	0.23

Figure 2. Space/Time for accessing array *LCP*. On the left for random accesses and on the right for sequential accesses.

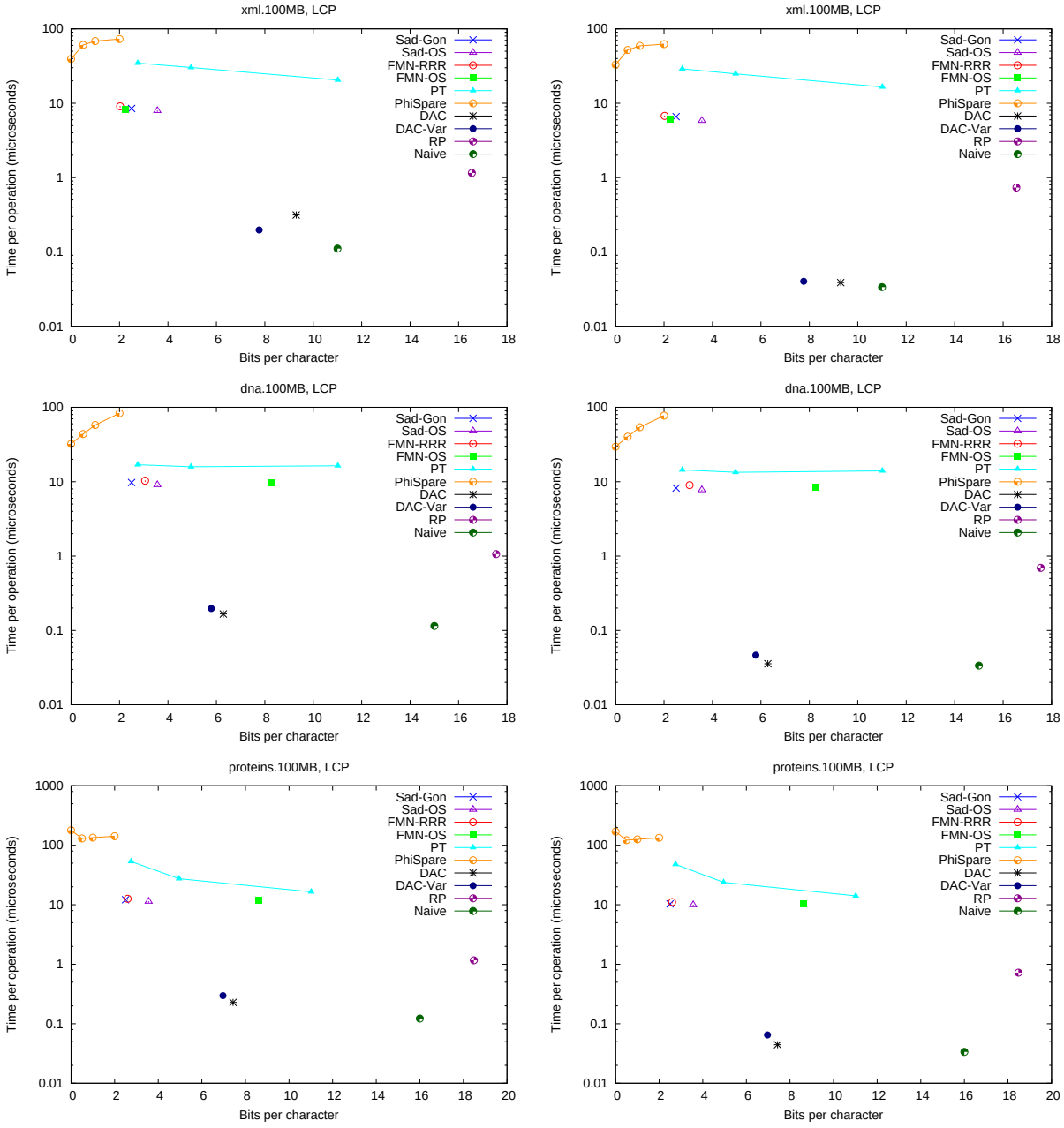
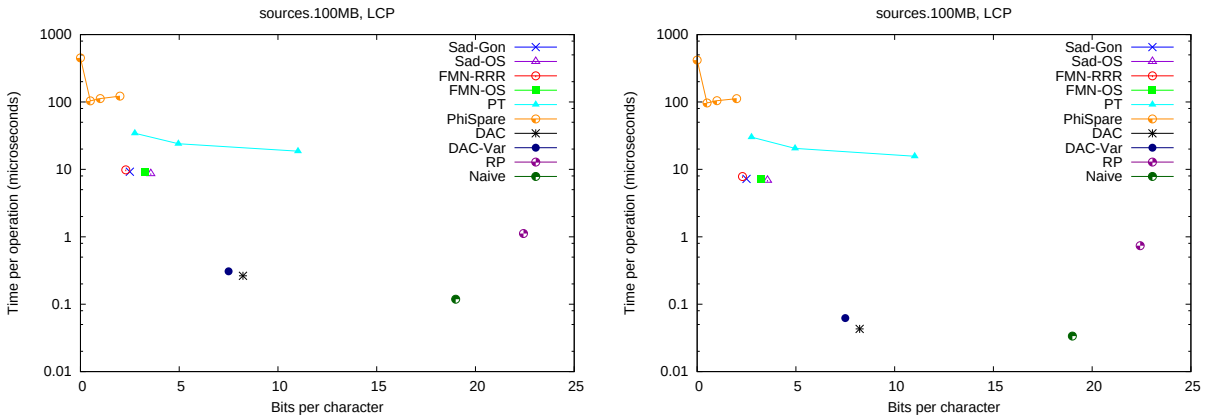


Figure 2. *Cont.*



We use Sadakane's CSA implementation [12], available at *Pizza & Chili*, as our CSA, with sampling $s = 16$. This is relevant because various LCP implementations need to compute $A[i]$ from the CSA.

A first observation is that Fischer *et al.*'s compression techniques for bitmap H do not compress significantly in practice (on non-repetitive collections). The value of r is not small enough to make the compressed representations noticeably smaller (and at times their space overhead over the entropy makes them larger, especially variant *FMN-OS*).

These explicit representations of H require only $2.1n$ bits of space and access *LCP* in about 10 microseconds. Technique *PT* is always dominated by them and *PhiSpare* can achieve less space, but at the price of an unacceptable 10-fold increase in time.

The other dominant technique is *DAC*/*DAC-Var*, which requires significantly more space ($6n$–$8n$ bits) but can access *LCP* within 0.1–0.2 microseconds. This is because it does not need to compute $A[i]$ to find $LCP[i]$. Its time is even competitive with *Naive*, while using much less space. Technique *RP*, instead, does not perform well, doing even worse than *Naive* in time and space. This shows, again, that (in non-repetitive) collections the value r is not small enough: While the length of the Re-Pair compressed sequence is 30%–60% of the original one, the values require more bits than in *Naive* because Re-Pair creates many new symbols.

DAC, *DAC-Var* and *Naive* are the only techniques that benefit from extracting consecutive values, so the former stay comparable with *Naive* in time even in this case. This owes to the better cache usage compared with the other representations, which must access spread text positions to decode a contiguous *LCP* area.

For the sequel we will keep only *DAC* and *DAC-Var*, which give the best time performance, and *FMN-RRR* and *Sad-Gon*, which have the most robust performance at representing H.

4. Computing *RMQ*, *PSV*, and *NSV*

Once a representation for *LCP* is chosen, one must carry out operations *RMQ*, *PSV*, and *NSV* on top of it (as they require to access *LCP*). We first implemented verbatim the theoretical proposals of Fischer *et al.* [18]. For *NSV*, the idea is akin to the recursive *findclose* solution for compressed trees [45]: The array is divided into blocks and some values are chosen as *pioneers* so that, if a position is not a pioneer, then its *NSV* answer is in the same block of that of its preceding pioneer (and thus it can be

found by scanning that block). Pioneers are marked in a bitmap so as to map them to a reduced array of pioneers, where the problem is recursively solved. We experimentally verified that it is convenient to continue the recursion until the end instead of storing the explicit answers at some point. The block length L yields a space/time tradeoff since, at each level of the recursion, we must obtain $O(L)$ values from *LCP*. *PSV* is symmetric, needing another similar structure.

For *RMQ* we apply a recent implementation [46,47] on the LCP array, which does not need to access *LCP* yet requires $2.44n$ bits. In the actual theoretical proposal [18] this space is reduced to $o(n)$ bits, but many accesses to *LCP* are necessary; we did not implement that idea verbatim as it has little chances of being practical.

The final data structure, which we call *FMN-NPR*, is composed of the structure to answer *NSV* plus the one for *PSV* plus the structure to calculate *RMQ*.

4.1. A Novel Practical Solution

We propose now a different solution, inspired in Sadakane and Navarro's *range min-max (RMM) tree* data structure [22]. We divide *LCP* into blocks of length L. Now we form a hierarchy of blocks, where we store the minimum *LCP* value of each block i in an array $m[i]$. The array uses $\frac{n}{L}\log n$ bits. On top of array m, we construct a perfect L-ary tree T_m where the leaves are the elements of m and each internal node stores the minimum of the values stored in its children. The total space for T_m is $\frac{n}{L}\log n(1+O(1/L))$ bits, so if $L=\omega(\log n)$, the space used is $o(n)$ bits.

To answer *NSV*(i), we look for the first $j>i$ such that *LCP*$[j]<p=$ *LCP*$[i]$, using T_m to find it in time $O(L\log(n/L))$. We first search sequentially for the answer in the same block of i. If it is not there, we go up to the leaf that represents the block and search the right siblings of this leaf. If some of these sibling leaves contain a minimum value smaller than p, then the answer to *NSV*(i) is within their block, so we go down to their block and find sequentially the leftmost position j where *LCP*$[j]<p$. If, however, no sibling of the leaf contains a minimum smaller than p, we continue going up the tree and consider the right siblings of the parent of the current node. At some node we find a minimum smaller than p and start traversing down the tree as before, finding at each level the first child of the current node with a minimum smaller than p. *PSV* is symmetric. As the minima in T_m are explicitly stored, the heaviest part of the cost in practice is the $O(L)$ accesses to *LCP* cells at the lowest levels.

To calculate *RMQ*(x,y) we use the same T_m and separate the search in three parts: (a) We calculate sequentially the minimum value in the interval $[x, L\lceil\frac{x}{L}\rceil-1]$ and its leftmost position in the interval; (b) We do the same for the interval $[L\lfloor\frac{y}{L}\rfloor, y]$; (c) We calculate *RMQ*$(L\lceil\frac{x}{L}\rceil, L\lfloor\frac{y}{L}\rfloor-1)$ using T_m. Finally we compare the results obtained in (a), (b) and (c) and the answer will be the one holding the minimum value, choosing the leftmost to break ties. For each node in T_m we also store the local position in the children where the minimum occurs, so we do not need to scan the child blocks when we go down the tree. The extra space incurred is just $\frac{n}{L}\log L(1+O(1/L))$ bits. The final data structure, if $L=\omega(\log n)$, requires $o(n)$ bits and can compute *NSV*, *PSV* and *RMQ* all using the same auxiliary structure. We call it *RMM-NPR*.

4.2. Experimental Comparison

We tested the performance of the different *NPR* implementations by performing 100,000 *NSV* and *RMQ* queries at random positions in the *LCP* array. Figure 3 shows the space/time achieved for each implementation on `dna` and `proteins` (the others texts gave very similar results). We consider the four selected LCP representations, which affect the performance because the algorithms have to access the array. The space we plot is additional to that to store the *LCP* array. We obtained space/time tradeoffs by using different block sizes $L = 8, 16, 32$. Note that *RMQ* on *FMN-RMQ* does not access *LCP*, so the value of L does not affect it.

Figure 3. Space/Time for the operations *NSV* and *RMQ*. Times for *PSV* are identical to those for *NSV*.

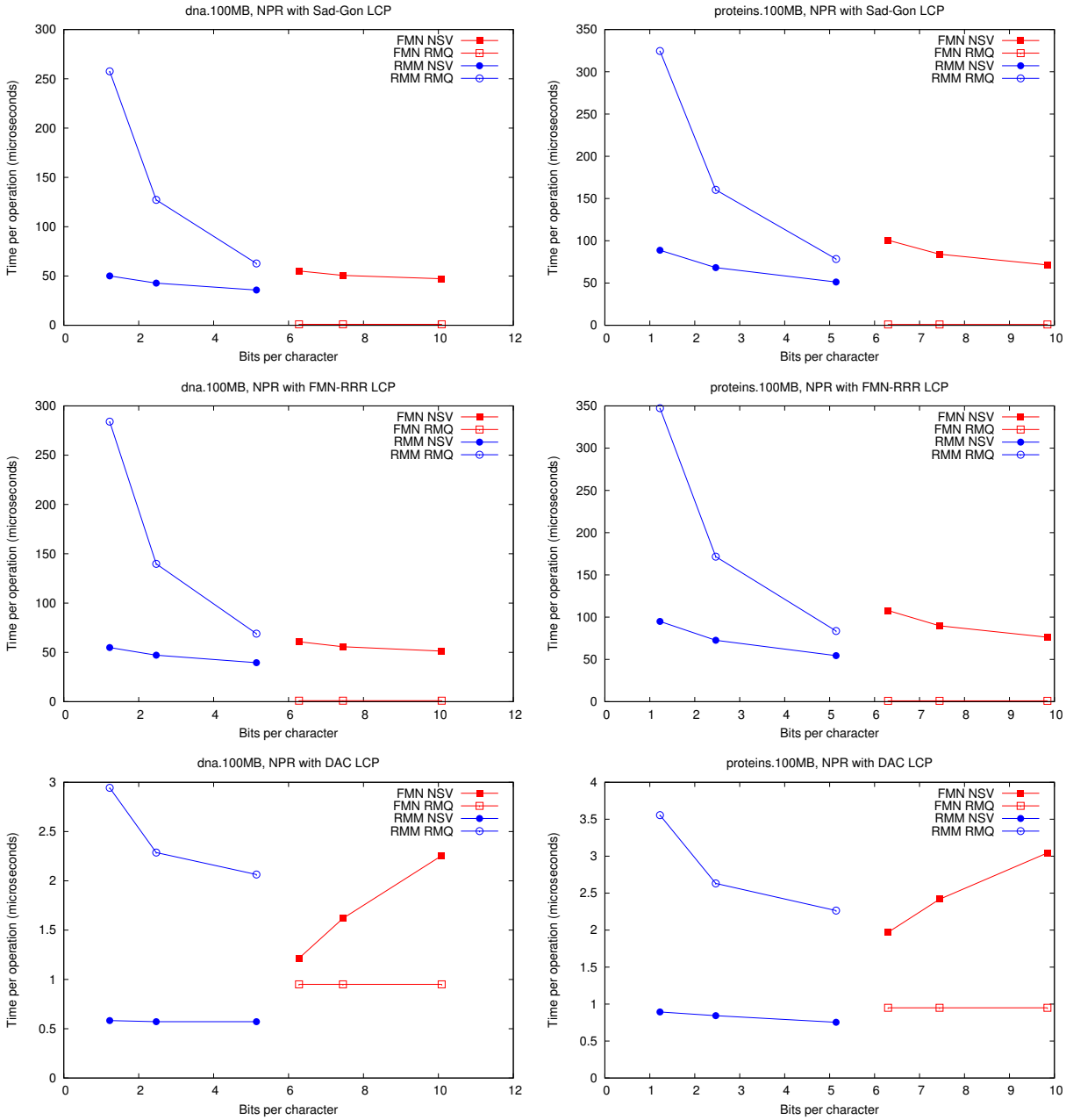
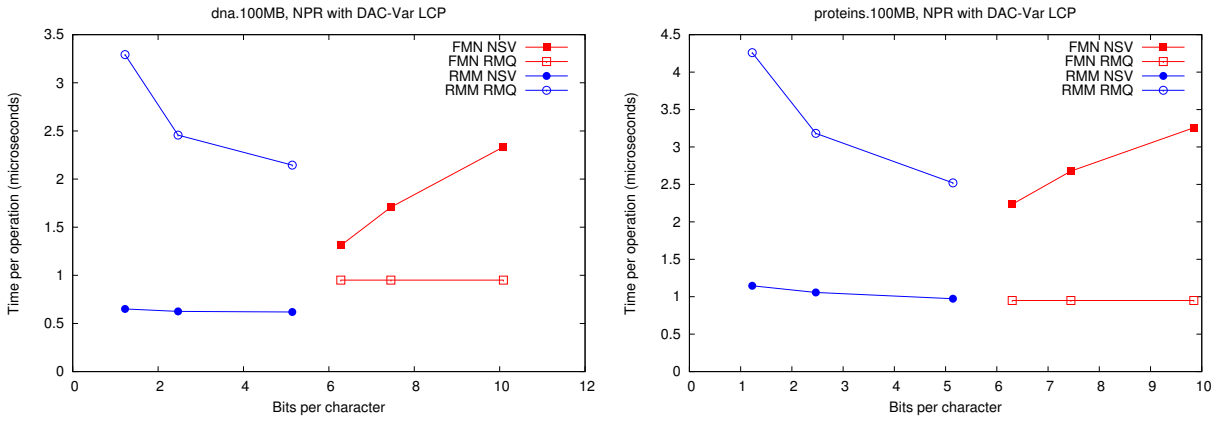


Figure 3. *Cont.*



First, it can be seen that *RMM-NPR* performs well using much less space than *FMN-NPR*. In addition, solving *NSV* (and thus *PSV*) with *RMM-NPR* structure is significantly faster than with *FMN-NPR*. This holds under all the *LCP* representations, regardless that times are one order of magnitude higher with representations *Sad-Gon* and *FMN-RRR*.

On the other hand, *RMQ* is way faster on *FMN* because it does not need to access *LCP* at all. It is 2–3 times faster than *RMM* using *DAC* or *DAC-Var* and two orders of magnitude faster than *RMM* using *Sad-Gon* or *FMN-RRR*. This operation, however, is far less common than *NSV* and *PSV* when implementing the CST operations, so its impact is not so high. Another reason to prefer *RMM-NPR* is that it can support a generalized version of *NSV* and *PSV*, which allows us to solve operation LAQ_T on suffix trees without adding more memory, as opposed to *FMN-NPR*. In the next section we show how a fully-functional CST can be implemented on top of the *RMM-NPR* representation.

5. Our Compressed Suffix Tree

Our CST implementation applies our *RMM-NPR* algorithms of Section 4 on top of some *LCP* representation from those chosen in Section 3. This solves most of the tree traversal operations by using the formulas provided by Fischer *et al*. [18], which we do not repeat here. In some cases, however, we have deviated from the theoretical algorithms for practical considerations.

***TDepth*:** We proceed by brute force using *Parent*, as there is no practical solution in the theoretical proposal.

***NSibling*:** There is a bug in the original formula [18] in the case v is the next-to-last child of its parent. According to them, $NSibling([v_l, v_r])$ first obtains its parent $[w_l, w_r]$, then checks whether $v_r = w_r$ (in which case there is no next sibling), then checks whether $w_r = v_r + 1$ (in which case the next sibling is leaf $[w_r, w_r]$), and finally answers $[v_r + 1, z - 1]$, where $z = RMQ(v_r + 2, w_r)$. This *RMQ* is aimed at finding the end of the next sibling of the next sibling, but it fails if we are near the end. Instead, we replace it by the faster $z = NSV'(v_r + 1, LCP[v_r + 1])$. $NSV'(i, d)$ generalizes *NSV* by finding the next value smaller or equal to d, and is implemented almost like *NSV* using T_m.

***Child*:** The children are ordered by letter. We need to extract the children sequentially using *FChild* and *NSibling*, to find the one descending by the correct letter, yet extracting the *Letter* of each is expensive. Thus we first find all the children sequentially and then binary search the correct letter among them, thus reducing the use of *Letter* as much as possible.

LAQ$_S(v,d)$**:** Instead of the slow complex formula given in the original paper, we use *NSV*′ (and *PSV*′): *LAQ*$_S([v_l,v_r],d) = [$*PSV*′$(v_l+1,d),$ *NSV*′$(v_r,d)-1]$. This is a complex operation we are supporting with extreme simplicity.

LAQ$_T(v,d)$**:** There is no practical solution in the original proposal. We proceed as follows to achieve the cost of d *Parent* operations, plus some *LAQ*$_S$ ones, all of which are reasonably cheap. Since *SDepth*$(v) \geq$ *TDepth*(v), we first try $v' =$ *LAQ*$_S(v,d)$, which is an ancestor of our answer; let $d' =$ *TDepth*(v'). If $d' = d$ we are done; else $d' < d$ and we try $v'' =$ *LAQ*$_S(v, d+(d-d'))$. We compute $d'' =$ *TDepth*(v'') (which is measured by using $d''-d'$ *Parent* operations until reaching v') and iterate until finding the right node.

Table 3 gives a space breakdown for our CST. We give the space in bpc of Sadakane's CSA, the 4 alternatives we consider for the LCP representation, and the space of our RMM tree (using $L = 32$ for a "slow" variant and $L = 16$ for a "fast" variant). The last column gives the total bpc considering the smallest "slow" (using *Sad-Gon* or *FMN-RRR* for *LCP*) and the smallest "fast" (using *DAC* or *DAC-Var*) alternatives.

Table 3. Space breakdown of our compressed suffix tree.

Collection	CSA bpc	LCP bpc				RMM bpc		Total bpc	
		Sad-Gon	*FMN-RRR*	*DAC*	*DAC-Var*	Slow	Fast	Slow	Fast
dna	5.88	2.10	3.06	6.29	5.79	1.23	2.47	9.21	14.14
xml	4.47	2.10	2.03	9.30	7.77	1.23	2.47	7.73	14.71
proteins	7.91	2.10	2.59	7.43	6.97	1.23	2.47	11.24	17.35
sources	5.20	2.10	2.29	8.23	7.50	1.23	2.47	8.53	15.17

6. Comparing the CST Implementations

We compare the following CST implementations: Välimäki *et al.*'s [20] implementation of Sadakane's compressed suffix tree [11] (*CST-Sadakane*); Russo's implementation of Russo *et al.*'s "fully-compressed" suffix tree [14] (*FCST*); and our best variants. These are called *Our CST* in the plots. Depending on their *LCP* representation, they are suffixed with *Sad-Gon*, *FMN-RRR*, *DAC*, and *DAC-Var*. We do not compare some operations like *Root* and *Ancestor* because they are trivial in all the implementations; *Locate* and *Count* because they depend only on the underlying CSA (which is mostly orthogonal, thus *Letter* is sufficient to study it); *SLink*^{i} because it is usually better to do *SLink* i times; and *LAQ*$_S$ and *LAQ*$_T$ because they are not implemented in the alternative CSTs (these two are shown only for ours).

We typically show space/time tradeoffs for all the structures, where the space is measured in bpc (recall that these CSTs replace the text, so this is the overall space required). The times are averaged

over a number of queries on random nodes. We use four types of node samplings, which make sense in different typical suffix tree traversal scenarios: (a) Collecting the nodes visited over 10,000 traversals from a random leaf to the root (used for *Parent*, *SDepth*, and *Child* operations); (b) Same but keeping only nodes with at least 5 children (for *Letter*); (c) Collecting the nodes visited over 10,000 traversals from the parent of a random leaf towards the root via suffix links (used for *SLink* and *TDepth*); and (d) Taking 10,000 random leaf pairs (for *LCA*). The standard deviation divided by the average is in the range [0.21, 2.56] for *CST-Sadakane*, [0.97, 2.68] for *FCST*, [0.65, 1.78] for *Our CST Sad-Gon*, [0.64, 2.50] for *Our CST FMN-RRR*, [0.59, 0.75] for *Our CST DAC*, and [0.63, 0.91] for *Our CST DAC-Var*. The standard deviation of the estimator is thus at most 1/100th of that.

***Parent* and *TDepth*.** Figure 4 shows the performance of operations *Parent* and *TDepth*. As can be seen, *CST-Sadakane* is about 5 to 10 times faster than our best results for *Parent* and 100 times faster for *TDepth*. This is not surprising given that *CST-Sadakane* stores the topology explicitly and thus these operations are among the easiest ones. In addition, our implementation computes *TDepth* by brute force. Note, on the other hand, that *FCST* is about 10 times slower than our slowest implementation.

Figure 4. Space/Time trade-off performance for operations *Parent* and *TDepth*. Note the log scale.

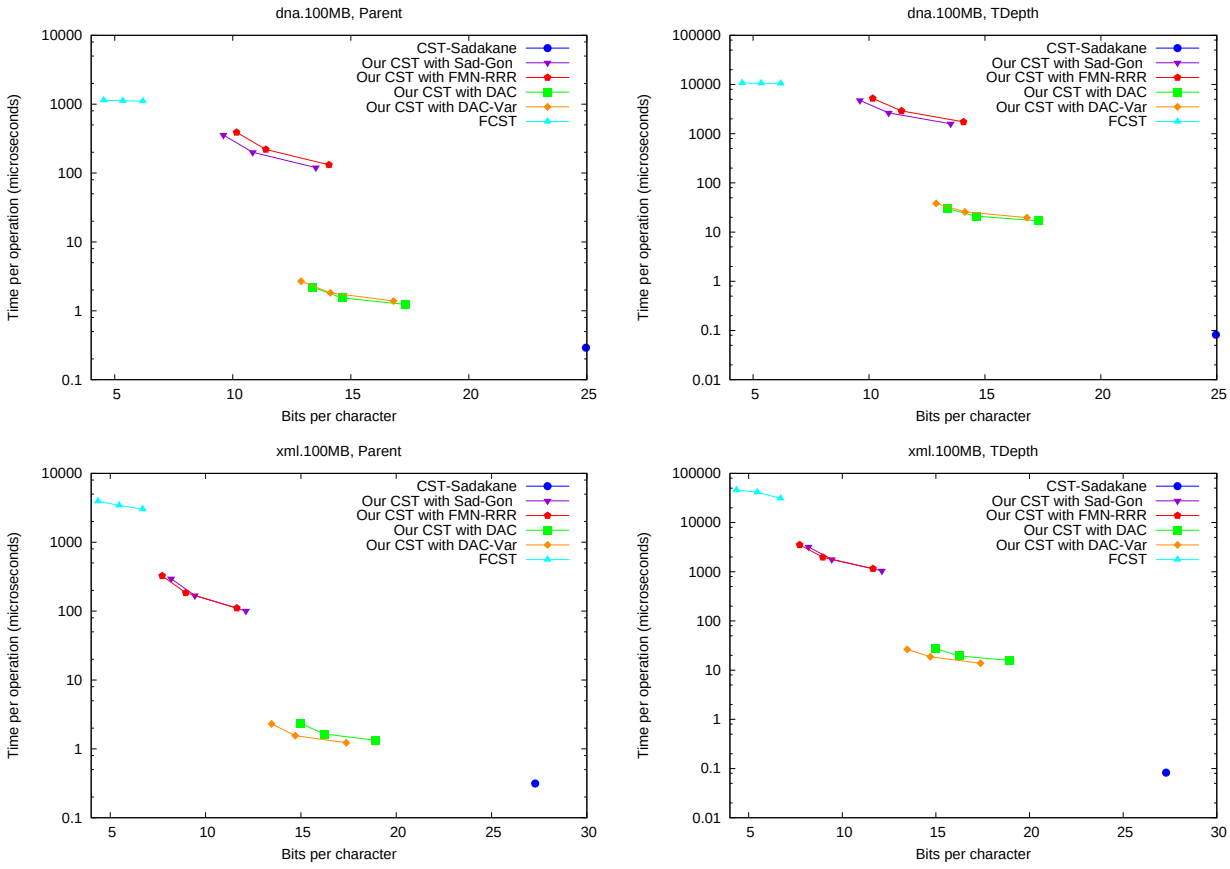
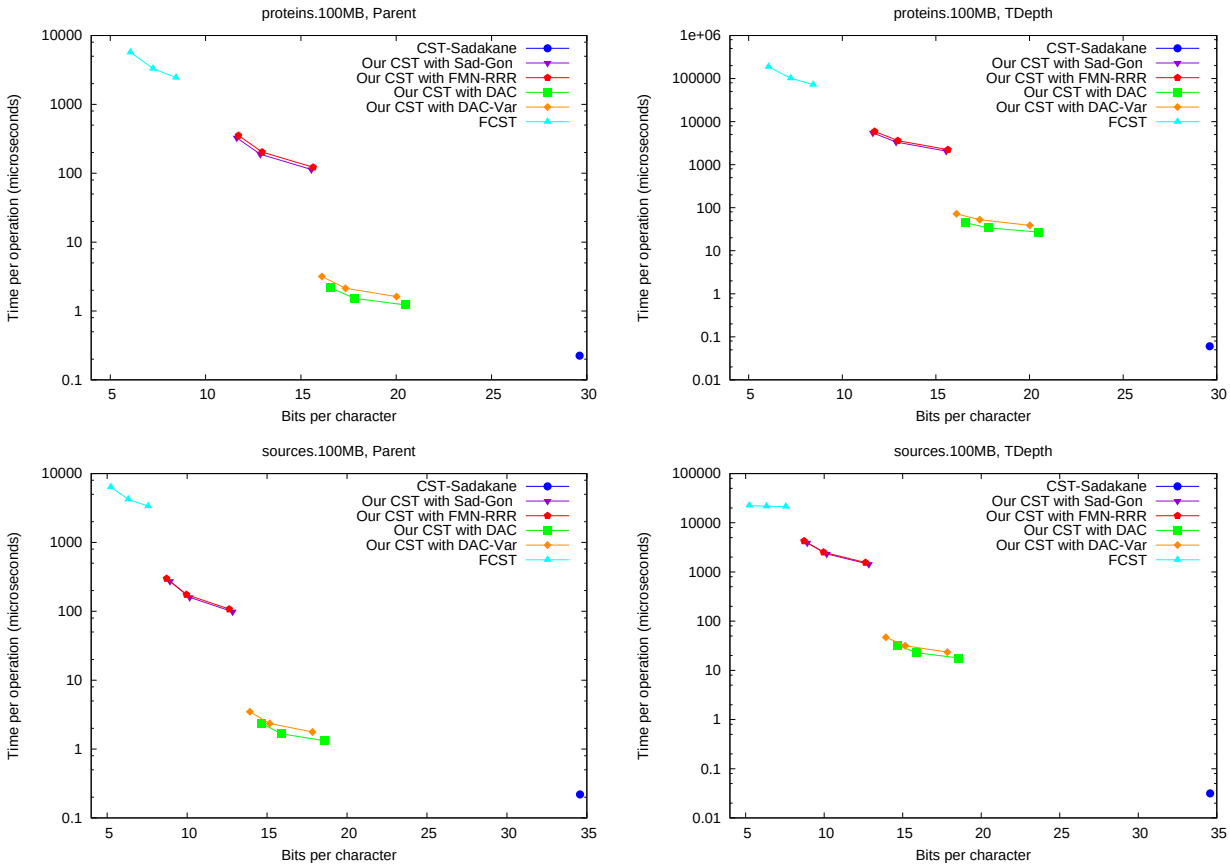


Figure 4. *Cont.*



***SLink*, *LCA*, and *SDepth*.** Figure 5 shows the performance of operations *SLink* and *LCA*, which are the ones most specific of suffix trees. Figure 6 (left) shows operation *SDepth*. Those operations happen to be among the most natural ones in Russo's *FCST*, which displays a better performance compared with the previous ones. *CST-Sadakane*, instead, is not anymore the fastest. Indeed, our implementations based on *DAC* and *DAC-Var* are by far the fastest ones for these operations, 10 times faster than *CST-Sadakane* and 100 to 1000 times faster than *FCST*. Even our slower implementations are way faster than *FCST* and close to *CST-Sadakane* times.

Figure 5. Space/Time trade-off performance for operations *SLink* and *LCA*. Note the log scale.

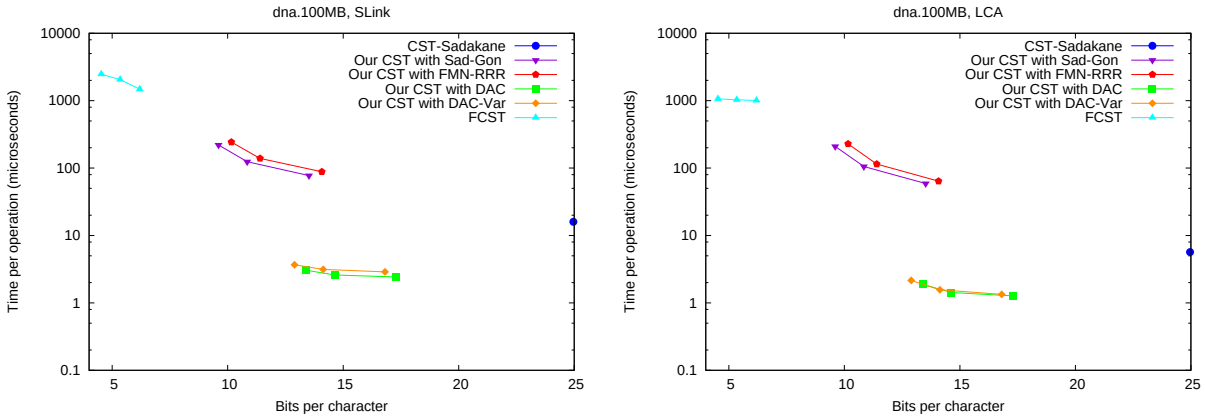
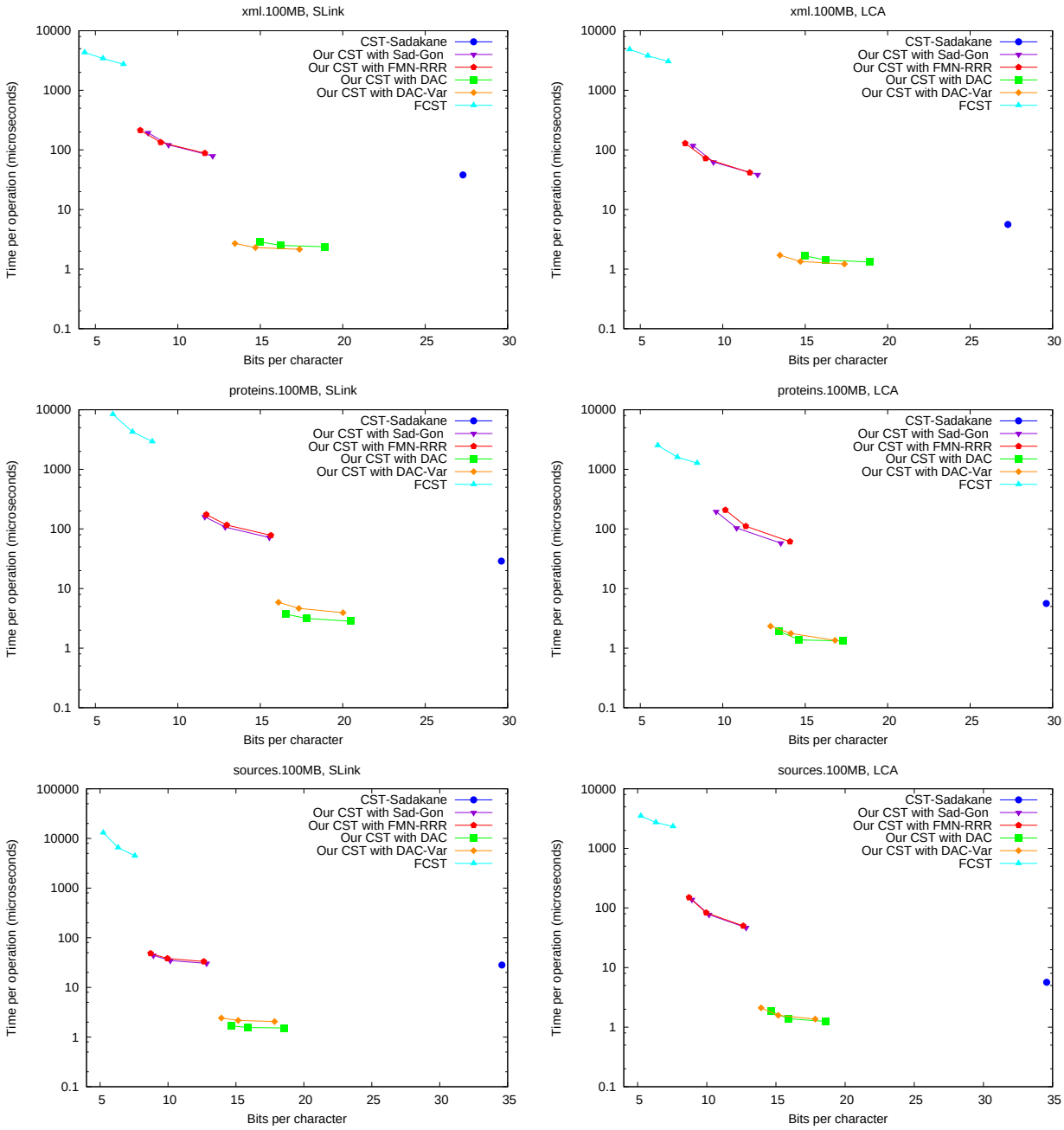
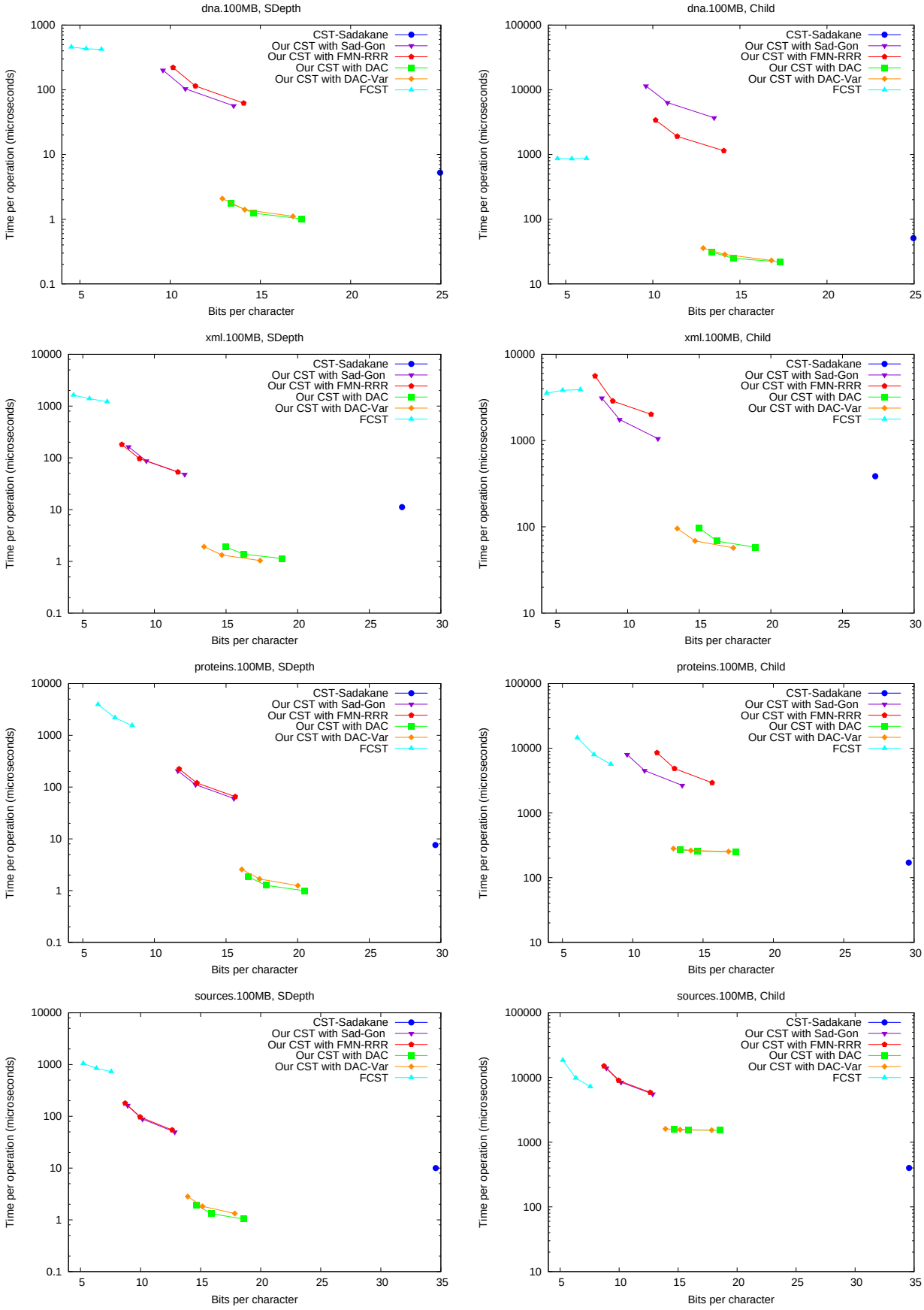


Figure 5. *Cont.*



Child. Figure 6 (right) shows operation *Child*, where we descend by a random child from the current node. Our times are significantly higher than for other operations, as expected, and the same happens to *CST-Sadakane*, which becomes comparable with our "large and fast" variants. *FCST*, on the other hand, is not much affected and becomes comparable with our "small and slow" variant. We note that, unlike other operations, the differences in performance depend markedly on the type of text.

Figure 6. Space/Time trade-off performance for operations *SDepth* and *Child*. Note the log scale.



Letter. Figure 7 (left) shows operation *Letter*(v, i), as a function of i. This operation depends only on the CSA structure, and requires either applying $i - 1$ times Ψ, or applying once A and A^{-1}. The former choice is preferred for *FCST* and the latter in *CST-Sadakane*. For our CST, using Ψ iteratively was better for these i values, as the alternative takes around 70 microseconds (recall we use Sadakane's CSA [12]).

Figure 7. Space/Time trade-off performance for operation *Letter* and for a full tree traversal. Note the log scale.

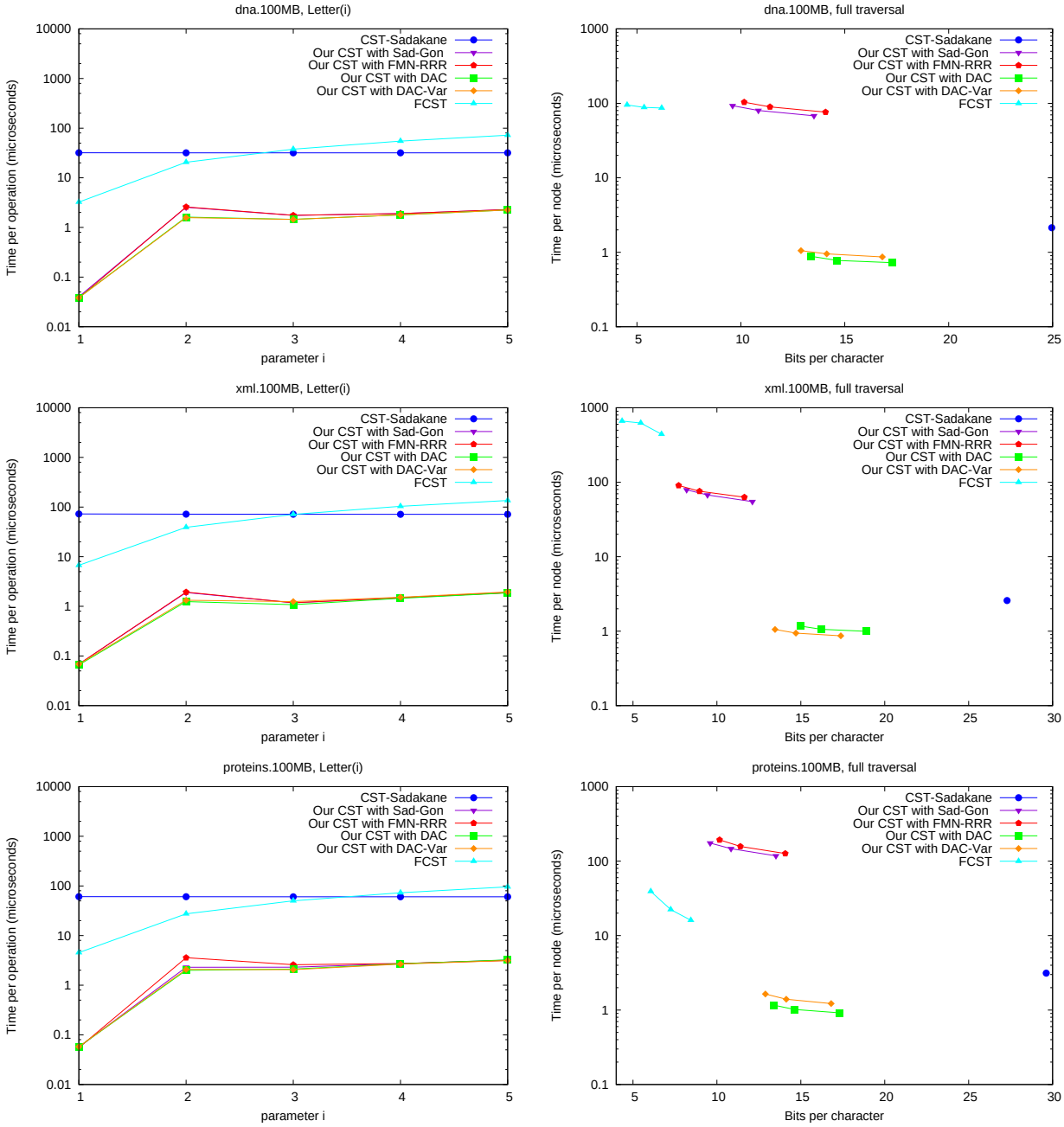
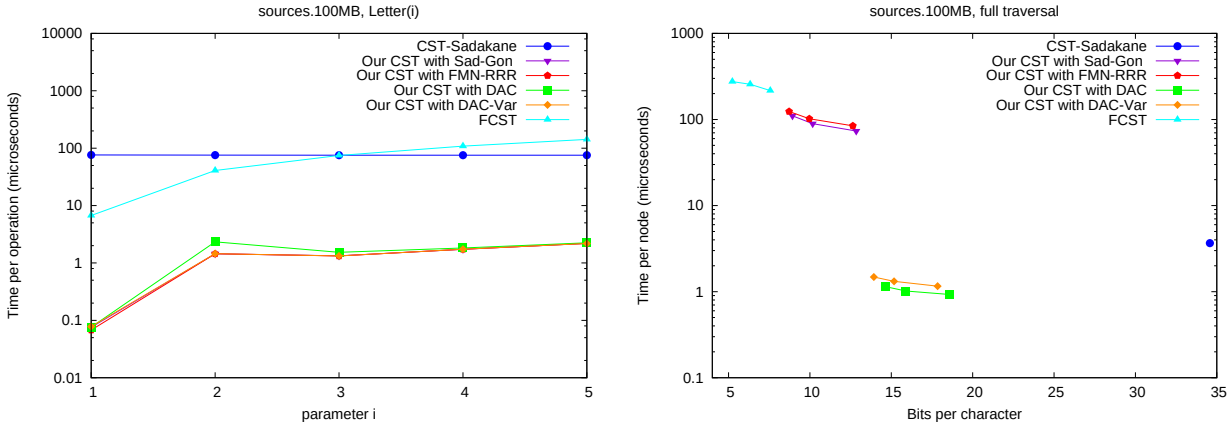


Figure 7. *Cont.*



A full traversal. Figure 7 (right) shows a basic suffix tree traversal algorithm: the classical one to detect the longest repetition in a text. This traverses all of the internal nodes using *FChild* and *NSibling* and reports the maximum *SDepth*. Although *CST-Sadakane* takes advantage of locality, our "large and fast" variants are better while using half the space. Our "small and slow" variant, instead, requires a few hundred microseconds, as expected, yet *FCST* has a special implementation for full traversals and it is competitive with our slow variant.

***LAQ*$_S$ and *LAQ*$_T$.** Figure 8 shows the performance of the operations $LAQ_S(v, d)$ and $LAQ_T(v, d)$. The times are presented as a function of parameter d, where we use $d = 4, 8, 16, 32, 64$. We query the nodes visited over 10,000 traversals from a random leaf to the root (excluding the root). As expected, the performance of LAQ_S is similar to that of *Parent*, and the performance of LAQ_T is close to that of doing *Parent* d times. Note that, as we increase the value of d, the times tend to a constant. This is because, as we increase d, a greater number of queried nodes are themselves the answer to LAQ_S or LAQ_T queries.

Figure 8. Space/Time trade-off performance for operation $LAQ_S(d)$ and $LAQ_T(d)$. Note the log scale.

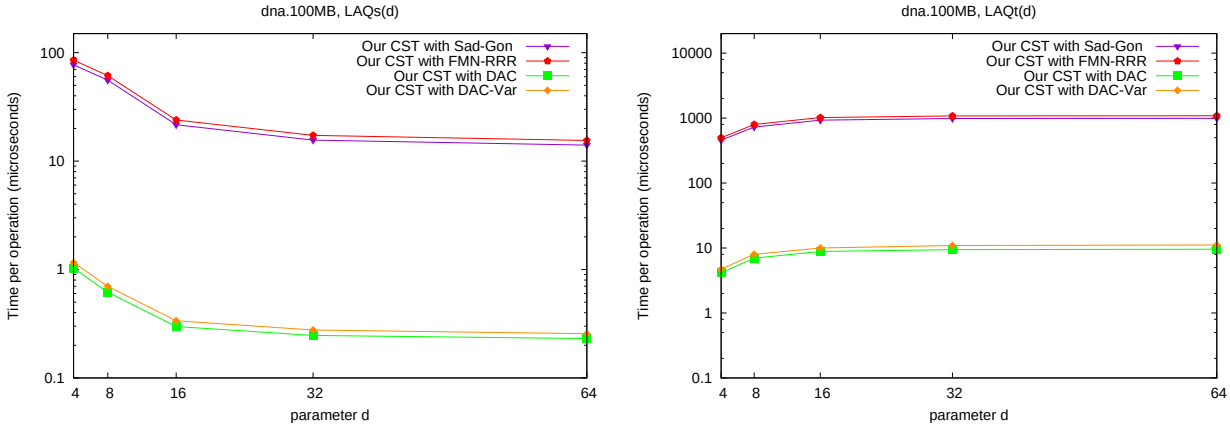
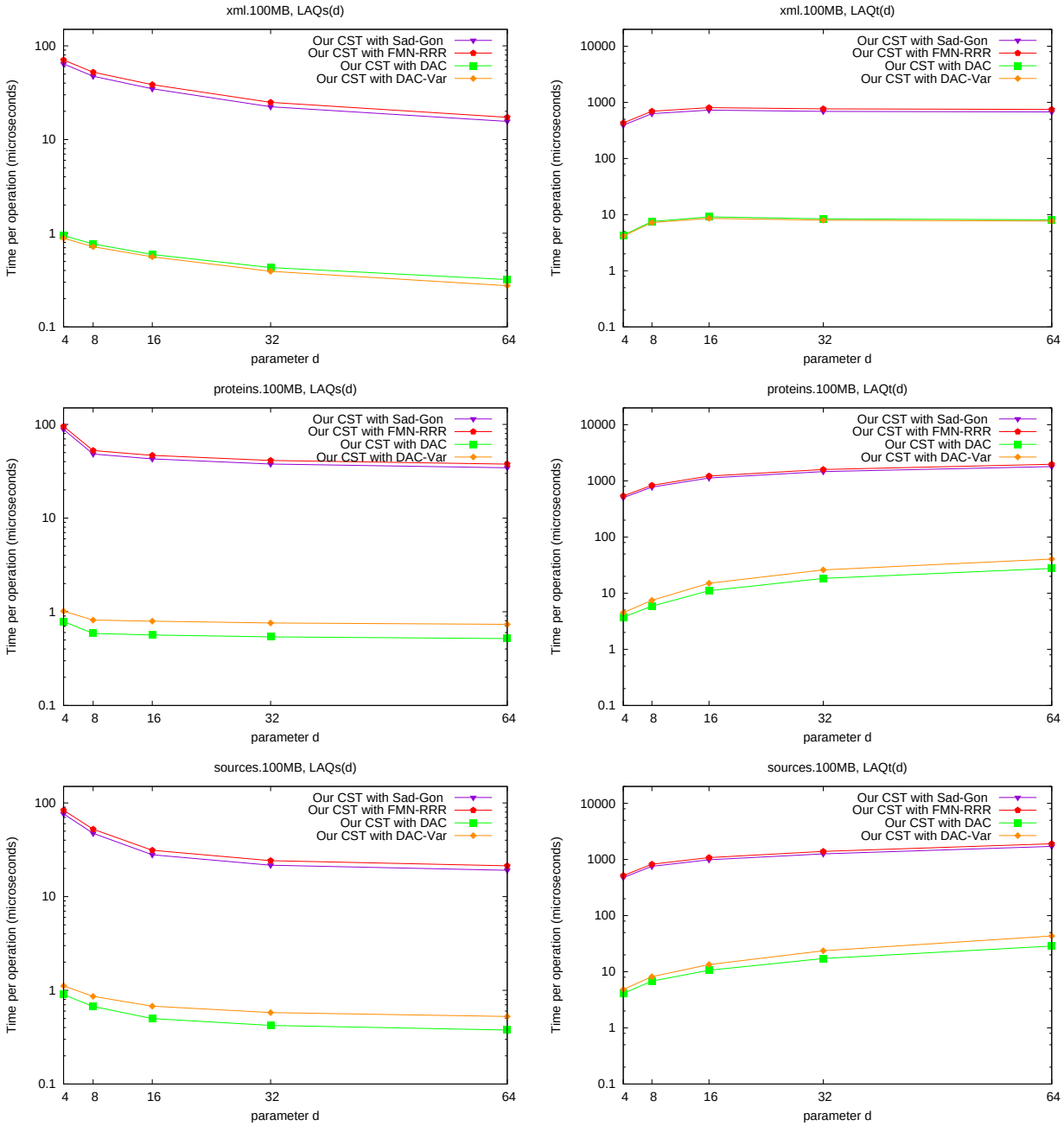


Figure 8. *Cont.*



Discussion. The general conclusion is that our CST implementation does offer a relevant tradeoff between the two rather extreme existing variants. Our CSTs can operate within 8–12 bpc (that is, at most 50% larger than the plain byte-based representation of the text, and replacing it) while requiring a few hundred microseconds for most operations (the "small and slow" variants *Sad-Gon* and *FMN-RRR*); or within 13–16 bpc and carrying out most operations within a few microseconds (the "large and fast" variants *DAC*/*DAC-Var*). In contrast, *FCST* requires only 4–6 bpc (which is, remarkably, as little as half the space required by the plain text representation), but takes the order of milliseconds per operation; and *CST-Sadakane* takes usually a few tens of microseconds per operation but requires 25–35 bpc, which is close to uncompressed suffix arrays (not to uncompressed suffix trees, though).

We remark that, for many operations, our "fast and large" variant takes half the space of *CST-Sadakane* and is many times faster. Exceptions are *Parent* and *TDepth*, where *CST-Sadakane* stores the explicit tree topology, and thus takes a fraction of a microsecond. On the other hand, our CST carries out LAQ_S in the same time of *Parent*, whereas this is much more complicated for the alternatives (they do not even implement it). For *Child*, where we descend by a random letter from the current node, the times are higher than for other operations, as expected, yet the same happens to all the implementations. We note that *FCST* is more efficient on operations *LCA* and *SDepth*, which are its kernel operations, yet it is still slower than our "small and slow" variant. Finally, *TDepth* is an operation where all but Sadakane's CST are relatively slow, yet on most suffix tree algorithms the string depth is much more relevant than the tree depth. Our $LAQ_T(v, d)$ costs about d times the time of our *TDepth*.

7. A Repetition-Aware CST

We now develop a variant of our *RMM* representation that is suitable for highly repetitive text collections, and upgrade it to a fully-functional CST for this case. Recall that we need three components: A CSA, an LCP representation, and fast *PSV*/*NSV*/*RMQ* queries on that *LCP* sequence.

We use the RLCSA [25] as the base CSA of our repetition-aware CST. We also use the compressed representation of *PLCP* (*i.e.*, bitmap H) [18]. Since now we assume $r \ll n$, we use a compressed bitmap representation that is useful for very sparse bitmaps [24]: We δ-encode the runs of 0's between consecutive 1's, and store absolute pointers to the representation of every sth 1. This is very efficient in space and solves $select_1$ queries in time $O(s)$, which is the operation needed to compute a *PLCP* value.

The main issue is how to support fast operations using the RLCSA and our LCP representation. Directly using the *RMM* tree structure requires at least 1 bpc (see Figure 3), which is too much when indexing repetitive collections. Our main idea is to replace the regular structure of tree T_m by the parsing tree obtained by a Re-Pair compression of the sequence *LCP*. We now explain this idea in detail.

7.1. Grammar-Compressing the LCP Array

To solve the queries on top of the LCP array, we represent a part of it in grammar-compressed form. This will be redundant with our compressed representation of array H, which was already explained.

Following the idea in Fischer *et al.* [18], we grammar-compress the differential LCP array, defined as $LCP'[i] = LCP[i] - LCP[i-1]$ if $i > 1$, and $LCP'[1] = LCP[1]$. This differential LCP array contains now $O(r)$ areas that are exact repetitions of others, and a Re-Pair-based compression of it yields $|R| + |C| = O(r \log \frac{n}{r})$ words [18,35]. We note, however, that the compression achieved in this way is modest [35]: We guarantee $O(r \log \frac{n}{r})$ *words*, whereas the RLCSA and PLCP representations require basically $O(r \log \frac{n}{r})$ *bits*. Indeed, the poor performance of variant *RP* in Section 3 shows that the idea should not be applied directly.

To overcome this problem, we take advantage of the fact that this representation is redundant with H. We *truncate* the parsing tree of the grammar, and use it as a device to speed up computations that would otherwise require expensive accesses to *PLCP* (*i.e.*, to bitmap H).

Let R and C be the results of compressing *LCP*′ with Re-Pair. Every nonterminal i of R expands to a substring $S[1, t]$ of *LCP*′. No matter where S appears in *LCP*′ (indeed, it must appear more than once),

we can store some values that are intrinsic to S. Let us define a relative sequence of values associated to S, as follows: $S'[0] = 0$ and $S'[j] = S[j] + S'[j-1]$. Then, we define the following variables associated to the nonterminal:

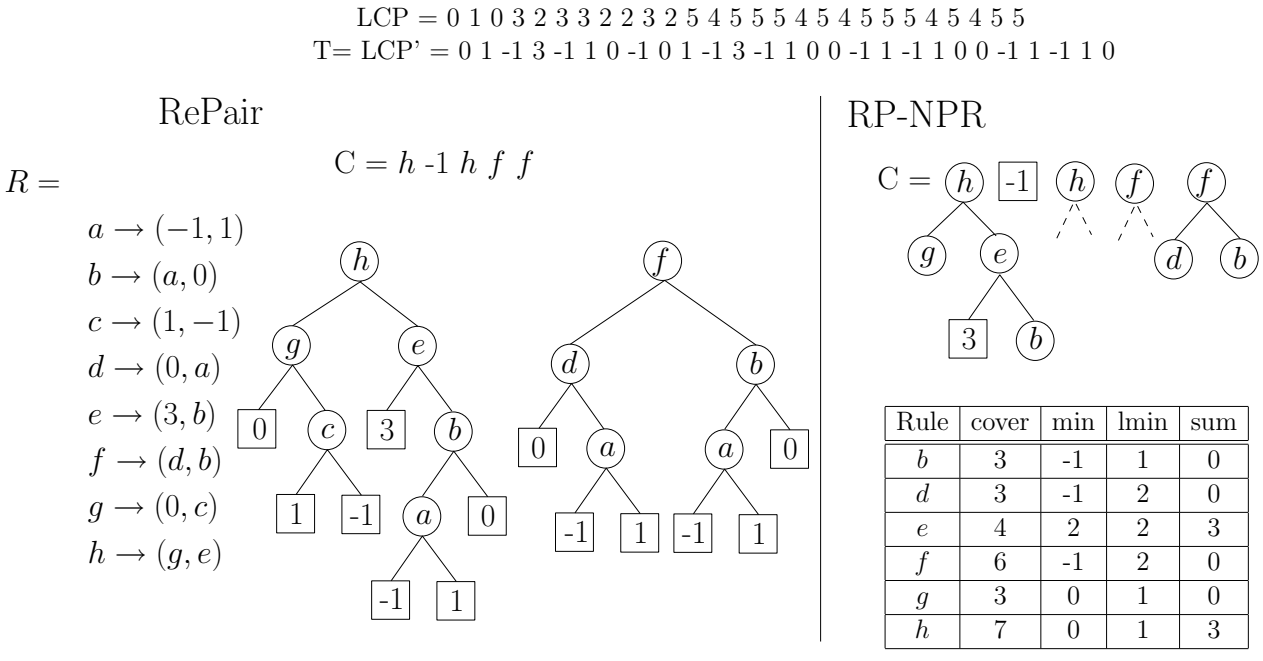
- $min_i = \min_{1 \le j \le t} S'[j]$ is the minimum value in S'.
- $lmin_i$ and $rmin_i$ are the leftmost and rightmost positions j where $S'[j] = min_i$, respectively.
- $sum_i = S'[t] = \sum_{1 \le j \le t} S[j]$ is the sum of the values $S[j]$.
- $cover_i = t$ is the number of values in S'.

As most of these values are small, we encode them with DACs [43], which use less space for short numbers while providing fast access ($rmin$ is stored as the difference with $lmin$).

To reduce space, we prune the grammar by deleting the nonterminals i such that $cover_i < t$, where t will be a space/time tradeoff parameter. However, "short" nonterminals that are mentioned in sequence C are not deleted.

This ensures that we can skip $\Omega(t)$ symbols of *LCP* with a single access to the corresponding nonterminal in C, except for the short nonterminals (and terminals) that are retained in C. To speed up traversals on C, we join together maximal consecutive subsequences of nonterminals and terminals in C that sum up a total $cover < t$: We create a new nonterminal rule in R (for which we precompute the variables above) and replace it in C, deleting those nonterminals that formed the new rule and do not appear anymore in C. This will also guarantee that no more than $O(t)$ accesses to *LCP* are needed to solve queries. Note that we could have built a hierarchy of new nonterminals by recursively grouping t consecutive symbols of C, achieving logarithmic operation times just as with tree T_m [29], but this turned out to be counterproductive in practice. Figure 9 gives an example.

Figure 9. On the left, example of the Re-Pair compression of a sequence T = *LCP'*. We show R in array form and also in tree form. On the right, our *RP-NPR* construction over T, pruning with $t = 4$. We show how deep can the symbols of C be expanded after the pruning.



Rule	cover	min	lmin	sum
b	3	-1	1	0
d	3	-1	2	0
e	4	2	2	3
f	6	-1	2	0
g	3	0	1	0
h	7	0	1	3

Finally, sampled pointers are stored to every c-th cell of C. Each sample for position $C[c \cdot j]$, stores:

- $Pos[j] = 1 + \sum_{1 \leq k \leq cj-1} cover_{C[k]}$, that is, the first position $LCP[i]$ corresponding to $C[c \cdot j]$.
- $Val[j] = \sum_{1 \leq k \leq cj-1} sum_{C[k]}$, that is, the value $LCP[i]$.

7.2. Computing NSV, PSV, and RMQ

To answer $NSV(i)$, we first look for the rule $C[j]$ that contains $LCP[i+1]$: We binary search Pos for the largest j' such that $Pos[j'] \leq i+1$ and then sequentially advance on $C[cj'..j]$ until finding the largest j such that $pos = Pos[j] + \sum_{cj' \leq k < j} cover_{C[k]} \leq i+1$. At the same time, we compute $\ell = Val[j'] + \sum_{cj' \leq k < j} sum_{C[k]}$.

Now, if $\ell + min_{C[j]} < LCP[i]$, it is possible that $NSV(i)$ is within the same $C[j]$. In this case, we search recursively the tree expansion with root $C[j]$ for the leftmost value to the right of i and smaller than $LCP[i]$: Let $C[j] \to ab$ in the grammar. We recursively visit child a if $\ell + min_a < LCP[i]$ and $pos + cover_a \geq i+1$. If we find no answer there, or we had decided not to visit a, then we set $\ell = \ell + sum_a$ and $pos = pos + cover_a$ and recursively visit child b if $\ell + min_b < LCP[i]$. Note that if $pos \geq i+1$ we do not need to enter b, but can simply use the value min_b and the position $pos + lmin_b$ for the minimum. If we also find no answer inside b, or we had decided not to visit b, we return with no value. On the other hand, if we reach a leaf l during the recursion, we sequentially scan the array $LCP[pos, pos + cover_l - 1]$, updating $\ell = \ell + LCP[k]$ and increasing pos. If at some position we find a value smaller than $LCP[i]$, we report the position pos.

If we return with no value from the first recursive call at $C[j]$, it was because the only values smaller than $LCP[i]$ were to the left of i. In this case, or if we had decided not to enter into $C[j]$ because $\ell + min_{C[j]} \geq LCP[i]$, we sequentially scan $C[j+1, n]$, while updating $\ell = \ell + sum_{C[k]}$ and $pos = pos + cover_{C[k]}$, until finding the first k such that $\ell + min_{C[k]} < LCP[i]$. Once we find such k, we are sure that the answer is inside $C[k]$. Thus we enter into $C[k]$ with a procedure very similar to the one for $C[j]$ (albeit slightly simpler as we know that all the positions are larger than i). In this case, as the LCP values are discrete, we know that if $\ell + min_{C[k]} = LCP[i] - 1$, there is no smaller value to the left of the min value, so in this case we directly answer the corresponding $lmin$ value, without accessing the LCP array. The solution to $PSV(i)$ is symmetric.

To answer $RMQ(x, y)$, we find the rules $C[i]$ and $C[j]$ containing x and y, respectively. We sequentially scan $C[i+1, j-1]$ and store the smallest $\ell + min_{C[k]}$ value found (in case of ties, the leftmost). If the minimum is smaller than the corresponding values $\ell + min_{C[i]}$ and $\ell + min_{C[j]}$, we directly return the value $pos + lmin_{C[k]}$ corresponding to position $C[k]$. Else, if the global minimum in $C[i]$ is equal to or less than the minimum for $i < k < j$, we must examine $C[i]$ to find the smallest value to the right of $x - 1$. Assume $C[i] \to ab$. We recursively enter into a if $pos + cover_a \geq x$, otherwise we skip it. Then, we update $\ell = \ell + sum_a$ and $pos = pos + cover_a$, and enter into b if $pos < x$, otherwise we directly consider $\ell + min_b$ as a candidate for the minimum sought. Finally, if we arrive at a leaf we scan it, updating ℓ and pos, and consider all the values ℓ where $pos \geq x$ as candidates to the minimum. The minimum for $C[i]$ is the smallest among all candidates to minimum considered, and with $pos + lmin_b$ or the leaf scanning process we know its global position. This new minimum is compared

with the minimum of $C[k]$ for $i < k < j$. Symmetrically, in case $k = j$ contains a value smaller than the minimum for $i \leq j < j$, we have to examine $C[j]$ for the smallest value to the left of $y + 1$.

7.3. Analysis

For the analysis we will assume that we hierarchically group the terminals and nonterminals of consecutive symbols in C (by extending the grammar) using a grouping factor of g, although, as explained, it turned out to be faster in practice not to do so. We will call h the height of the grammar, which can be made $O(\log n)$ by generating balanced grammars [48] (we experiment with balanced and unbalanced grammars in the next section). We also use sampling steps c to access C and s for the *PLCP* representation. We also assume s is the sampling step of the RLCSA, thus accessing one *PLCP* cell takes $O(s \log n)$ time.

Let us first consider operation *NSV* (and *PSV*). We spend $O(\log n + c)$ time to binary search *Pos* and move from $C[cj']$ to $C[j]$. If we have to expand $C[j]$, the cost increases by $O(h)$: Note that if we had entered a, and obtained no answer inside it, then we can directly use min_b and thus do not enter b, so we only enter one of the two. At the leaf of the grammar tree of $C[j]$, we may scan $O(t)$ symbols of *LCP* using *PLCP*, which costs $O(ts \log n)$. Further, the scanning of $C[j + 1, n]$ takes time $O(g \log n)$ assuming the hierarchical grouping, and the final expansion of $C[k]$ takes again $O(h + ts \log n)$ time.

Now consider operation *RMQ*. Again, we spend $O(c + \log n)$ time to find $C[i]$ and $C[j]$, and then expand them as before: Only one of the two branches is expanded in each case, so the time is $O(h + ts \log n)$. Finally, we need time $O(g \log n)$ to traverse the area $C[i + 1, j - 1]$.

Thus, for all the operations, the cost is $O(c + h + (ts + g) \log n)$.

Let $r' = |R| + |C|$, and recall that r is the number of runs, so $n' = O(r \log \frac{n}{r})$ if using Re-Pair. The sampling parameters involve $O((n'/c) \log n')$ bits for the sampling of C, plus $O((r/s) \log n)$ for the sampling of *PLCP*, plus $O((n/s) \log n)$ for the sampling of the RLCSA, plus $O((n'/g) \log n')$ for the new rules created in the hierarchy for C. Parameter t involves in practice an important reduction in $|R|$, the set of rules, but this cannot be upper bounded in the worst case unless the grammar is balanced, in which case it is reduced to $O(|R|/t)$ rules.

By choosing $c = g = \log^3 n$, $s = \log^2 n$, $t = \log n$, and assuming a balanced grammar, the cost of the operations simplifies to $O(\log^4 n)$ and the extra space incurred by the samplings is $o(r) + O(n/\log n)$ bits. This can be reduced to any $o(r) + O(n/\log^k n)$, for any constant k, by raising the time complexity to $O(\log^{k+3} n)$.

8. Experimental Evaluation on Repetitive Collections

We used various DNA collections from the Repetitive Corpus of *Pizza & Chili* [49]. We took DNA collections `Influenza` and `Para`, which are the most repetitive ones, and `Escherichia`, a less repetitive one. These are collections of genomes of various bacteria of those species. We also use `Plain DNA`, which is plain DNA from *PizzaChili*, as a non-repetitive baseline. On the other hand, in order to show how results scale with repetitiveness, we also included `Einstein`, corresponding to the Wikipedia versions of the article about Einstein in German, although it is not a biological collection.

Table 4 (left) gives some basic information on the collections, including the number of runs in Ψ (much lower than those for general sequences; see Table 2).

For the RLCSA we used a fixed sampling that gave reasonable performance: One absolute value out of 32 is stored to access $\Psi(i)$, and one text position every 128 is sampled to compute $A[i]$. Similarly, we used sampling step 32 for the δ-encoding of the bitmaps Z and O that encode H.

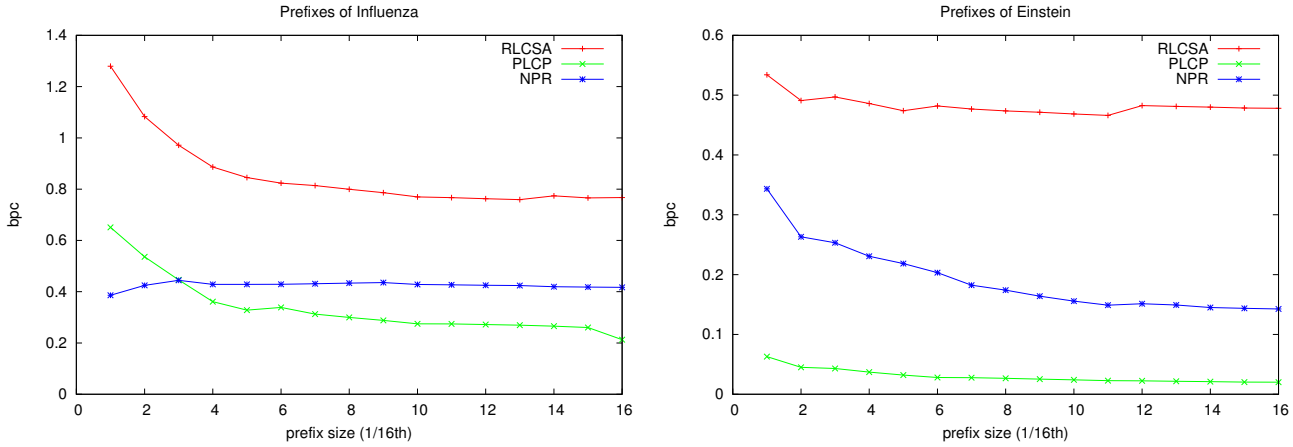
Table 4. Text size and size of our CST (which replaces the text), both in MB, and number of runs in Ψ. On the right, bpc for the different components, and total bpc of the different collections considered. The NPR structure is the smallest setting between *RP-NPR* and *RPBal-NPR* for that particular text.

Collection	Text Size	Runs (r/n)	CST size	RLCSA	(P)LCP	NPR	Total
`Influenza`	148	0.019	27	0.77	0.21	0.42	1.40
`Para`	410	0.036	67	1.28	0.26	0.36	1.90
`Escherichia`	108	0.134	48	2.46	0.92	0.39	3.77
`Plain DNA`	50	0.632	61	5.91	3.62	0.29	9.82
`Einstein`	89	0.001	3	0.48	0.01	0.14	0.63

Table 4 (right) shows the resulting sizes. The bpc of the CST is partitioned into those for the RLCSA, for the PLCP (H), and for NPR, which stands for the data structure that solves *NSV*/*PSV*/*RMQ* queries. For the latter we used $t = 256$, which offered answers within 2 milliseconds (ms). Note that Fischer *et al.*'s [18] compression of bitmap H does work in repetitive sequences, unlike what happened in the general case: We reduce the space from 2 bpc to 0.21–0.26 bpc on the more repetitive DNA sequences, for example. Similarly, the regular RMM-tree uses 1–5 bpc, whereas our version adapted to repetitive sequences uses as little as 0.3–0.4 bpc. The good space performance of the RLCSA, compared with a regular CSA for general sequences, also contributes to obtain a remarkable final space of 1.4–1.9 bpc for the more repetitive DNA sequences. This value deteriorates until approaching, for plain (non-repetitive) DNA, the same 10 bpc that we obtained before. Thus our data structure adapts smoothly to the repetitiveness (or lack of it) of the collection.

On the other hand, on Einstein, which is much more repetitive, the total space gets as low as 0.6 bpc. This is a good indication of what we can expect on future databases with thousands of individuals of the same species, as opposed to these testbeds with a few tens of individuals, or with more genetic variation. To give more insight into how the space of our structures evolve as the repetitiveness increases, Figure 10 shows the space used by each component of the index on increasing prefixes of collections `Influenza` and `Einstein`. It can be seen that, as the we index more of the collection, the space of the different components decreases until reaching a stable point that depends on the intrinsic repetitiveness of the collection (this is reached earlier for `Influenza`). Note, however, that even in the more repetitive `Einstein`, the RLCSA stabilizes very early at a fixed space. This space is that of the suffix array sampling, which is related to the speed of computing the contents of suffix array cells (and hence computing LCP values). It is interesting that Mäkinen *et al.* [25] proposed a solution to compress this array that proved impractical for the small databases we are experimenting with, but whose asymptotic properties ensure that will become practical for sufficiently large and repetitive collections.

Figure 10. Evolution of the space, in bpc, of our structures for increasing prefixes of collections `Influenza` and `Einstein`.



Let us discuss the NPR operations now. We used a public Re-Pair compressor built by ourselves [50], which offers two alternatives when dealing with symbols of the same frequencies. The basic one, which we will call *RP-NPR*, stacks the pairs of the same frequency, whereas the second one, *RPBal-NPR*, enqueues them and obtains much more balanced grammars in practice. For our structure we tested values $t = c = 64, 128, 256, 512$. We also include the basic regular RMM-tree of the previous sections (but running over our RLCSA and PLCP representations), to show that on repetitive collections our grammar-based version offers better space/time tradeoffs than the regular tree T_m. For this version, *RP-RMM*, we used values $L = 36, 64, 128, 256, 512$.

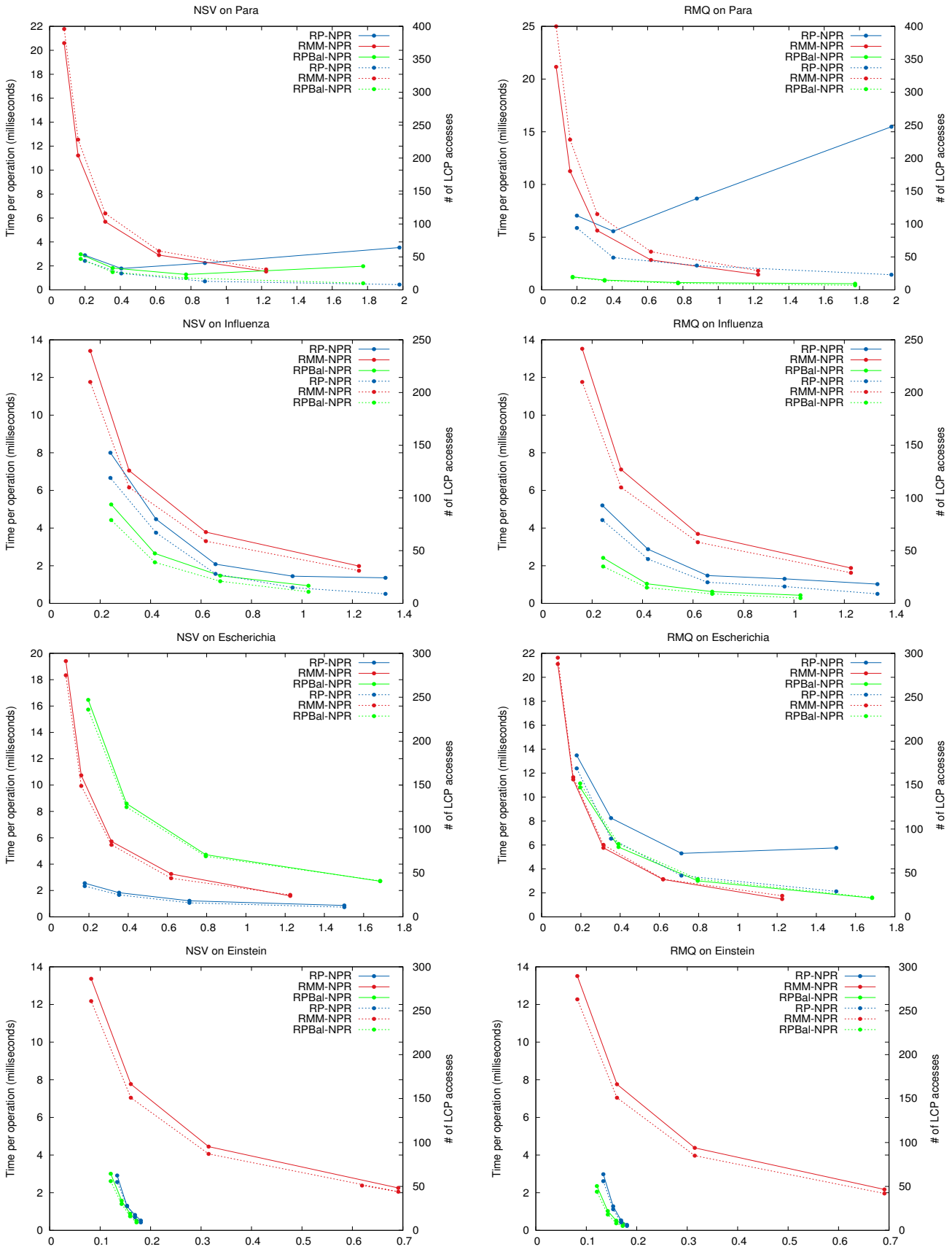
We measure the times of operations *NSV* (as *PSV* is symmetric) and *RMQ* following the methodology of Section 6: We choose 10,000 random suffix tree leaves (corresponding to uniformly random suffix array intervals $[v_l, v_r] = [v, v], v \in [1, n]$) and navigate towards the root using operation $\mathit{Parent}(v_l, v_r) = [\mathit{PSV}(v_l), \mathit{NSV}(v_r) - 1]$. At each such node, we also measure the string depth, $\mathit{SDepth}(v_l, v_r) = \mathit{LCP}[\mathit{RMQ}(v_l + 1, v_r)]$. We average the times of the *NSV* and *RMQ* queries involved.

Figure 11 shows the space/time performance of *RMM-NPR*, *RP-NPR*, and *RPBal-NPR* on the repetitive collections. In addition, it shows the number of explicit accesses to *LCP* made per NPR operation.

In general the curves for total time and number of *LCP* accesses have the same shape, showing that in practice the main cost of the NPR operations lies in retrieving the LCP values. In some cases, and most noticeably for *RMQ* queries on `Para` using *RP-NPR*, using more space yields fewer *LCP* accesses but higher time. This is because the Re-Pair grammars may be very unbalanced, and giving more space to them results in less tree pruning. Thus deeper trees, whose nodes have to be traversed one by one to cover short *LCP* passages, are generated. *RPBal-NPR* largely corrects this effect by generating more balanced grammar trees.

Clearly, *RP-NPR* and *RPBal-NPR* dominate the space/time map for all queries. They always make better use of the space than the regular tree of *RMM-NPR*. *RPBal-NPR* is usually better than *RP-NPR*, except on some particular cases, like *NSV* on `Escherichia`, where *RP-NPR* is faster.

Figure 11. Space/Time performance of NPR operations. The x-axis shows the size of the NPR structure. On the left y-axis the average time per operation (solid lines), and on the right y-axis the average number of LCP accesses per operation (dashed lines). Note the log scale on y for `Einstein`.



The times of our best variants are in the range of a few milliseconds per operation. This is high compared with the times obtained on general sequences. The closest comparison point is Russo *et al.*'s CST [14] whose times are very similar, but it uses 4–6 bpc compared with our 1.4–1.9 bpc. As discussed, we can expect our bpc value to decrease even more on larger databases. Such a representation, even if taking milliseconds per operation, is very convenient if it saves us from using a disk-based suffix tree. Note also that, on very repetitive collections, we could avoid pruning the grammar at all. In this case the grammar itself would give access to the LCP array much faster than by accessing bitmap H.

9. Conclusions

We have presented new practical compressed suffix tree implementations that offer very relevant space/time tradeoffs. This opens the door to a number of practical suffix tree applications, particularly relevant to bioinformatics. We have left the code publicly available at the *Pizza & Chili* site, to foster its widest dissemination [51].

Our main idea is to adapt range min-max trees [22] to the problem of solving queries on the LCP array. This has proved to offer an attractive combination of time and space. Moreover, it offers stronger functionality, allowing us to easily implement complex operations like string-level ancestors (LAQ_S), which are seldom implemented in other CSTs. The second key idea is that, on repetitive collections, one can grammar-compress the differential LCP array and use the grammar tree as a (non-regular) variant of the range min-max tree, whose size depends on the repetitiveness of the text collection. This has yielded the first compressed suffix tree that takes full advantage of repetitive collections, offering spaces well below 2 bits per character.

This is an active and lively research topic, and there has been interesting advances since our first publication [29]. Gog [21] developed another LCP representation that falls between *DAC*/*DAC-Var* and *Sad-Gon*/*FMN-RRR* (recall Section 3). Ohlebusch *et al.* [21,28] developed a data structure using $3n + o(n)$ bits that answers the three queries *NSV*/*PSV*/*RMQ* efficiently (recall Section 4). Both techniques were combined to build an alternative CST implementing Fischer *et al.*'s [18] idea, which outperforms our "large and fast" variant based on *DAC*/*DAC-Var*. Our "small and slow" variant, instead, still requires less space, and in this aspect is only outperformed by Russo *et al.*'s [14] CST, which is orders of magnitude slower. Furthermore, our technique is unbeaten when adapted to repetitive collections, which none of the other existing CSTs exploits well. As an example, our adapted range min-max tree uses 0.2–0.3 bpc on these collections, which is at least 10–15 times smaller than the new $3n + o(n)$ *NSV*/*PSV*/*RMQ* structure of Ohlebusch *et al.* [28].

Acknowledgements

We thank Francisco Claude, Johannes Fischer, Rodrigo González, Juha Kärkkäinen, Roberto Konow, Susana Ladra, Veli Mäkinen, Simon Puglisi, Luís Russo, and Kunihiko Sadakane for code, help to use it, and discussions. This work was partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. MIT Technology Review. Available online: http://www.technologyreview.com/featuredstory/511051/inside-chinas-genome-factory (accessed on 20 May 2013).
2. McCreight, E. A space-economical suffix tree construction algorithm. *J. ACM* **1976**, *32*, 262–272.
3. Weiner, P. Linear Pattern Matching Algorithms. In Proceedings of the IEEE Symposium on Switching and Automata Theory, Iowa City, IA, USA, 15–17 October 1973; pp. 1–11.
4. Apostolico, A. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*; Springer-Verlag: Berlin, Germany, 1985; pp. 85–96.
5. Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*; Cambridge University Press: Cambridge, UK, 1997.
6. Kurtz, S. Reducing the space requirements of suffix trees. *Softw. Prac. Exp.* **1999**, *29*, 1149–1171.
7. Manber, U.; Myers, E. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* **1993**, *22*, 935–948.
8. Abouelhoda, M.; Kurtz, S.; Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *J. Discret. Algorithms* **2004**, *2*, 53–86.
9. Munro, I.; Raman, V.; Rao, S. Space efficient suffix trees. *J. Algorithms* **2001**, *39*, 205–222.
10. Sadakane, K. Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays. In Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), San Francisco, CA, USA, 6–8 January 2002; pp. 225–232.
11. Sadakane, K. Compressed suffix trees with full functionality. *Theory Comput. Syst.* **2007**, *41*, 589–607.
12. Sadakane, K. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms* **2003**, *48*, 294–313.
13. Russo, L.; Navarro, G.; Oliveira, A. Fully-Compressed Suffix Trees. In Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN), Rio de Janeiro, Brazil, 7–11 April 2008; pp. 362–373.
14. Russo, L.; Navarro, G.; Oliveira, A. Fully-compressed suffix trees. *ACM Trans. Algorithms* **2011**, *7*, doi:10.1145/2000807.2000821.
15. Ferragina, P.; Manzini, G.; Mäkinen, V.; Navarro, G. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* **2007**, *3*, doi:10.1145/1240233.1240243.
16. Manzini, G. An analysis of the Burrows-Wheeler transform. *J. ACM* **2001**, *48*, 407–430.
17. Fischer, J.; Mäkinen, V.; Navarro, G. An(other) Entropy-Bounded Compressed Suffix Tree. In Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM), Pisa, Italy, 18–20 June 2008; pp. 152–165.
18. Fischer, J.; Mäkinen, V.; Navarro, G. Faster entropy-bounded compressed suffix trees. *Theory Comput. Sci.* **2009**, *410*, 5354–5364.
19. Fischer, J. Wee LCP. *Inf. Process. Lett.* **2010**, *110*, 317–320.
20. Välimäki, N.; Gerlach, W.; Dixit, K.; Mäkinen, V. Engineering a Compressed Suffix Tree Implementation. In Proceedings of the 6th Workshop on Experimental Algorithms (WEA), Rome, Italy, 6–8 June 2007; pp. 217–228.

21. Gog, S. Compressed Suffix Trees: Design, Construction, and Applications. Ph.D. Thesis, University of Ulm, Ulm, Germany, 2011.
22. Sadakane, K.; Navarro, G. Fully-Functional Succinct Trees. In Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Austin, TX, USA, 17–19 January 2010; pp. 134–149.
23. Arroyuelo, D.; Cánovas, R.; Navarro, G.; Sadakane, K. Succinct Trees in Practice. In Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX), New York, NY, USA, 3 January 2009; pp. 84–97.
24. Kreft, S.; Navarro, G. Self-Indexing Based on LZ77. In Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM), Palermo, Italy, 27–29 June 2011; pp. 41–54.
25. Mäkinen, V.; Navarro, G.; Sirén, J.; Välimäki, N. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.* **2010**, *17*, 281–308.
26. Claude, F.; Navarro, G. Self-Indexed Text Compression Using Straight-Line Programs. In Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS), Novy Smokovec, Slovakia, 24–28 August 2009; pp. 235–246.
27. Claude, F.; Fariña, A.; Martínez-Prieto, M.; Navarro, G. Compressed q-Gram Indexing for Highly Repetitive Biological Sequences. In Proceedings of the 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE), Philadelphia, PA, USA, 31 May–3 June 2010; pp. 86–91.
28. Ohlebusch, E.; Fischer, J.; Gog, S. CST++. In Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE), Los Cabos, Mexico, 11–13 October 2010; pp. 322–333.
29. Cánovas, R.; Navarro, G. Practical Compressed Suffix Trees. In Proceedings of the 9th International Symposium on Experimental Algorithms (SEA), Ischia Island, Italy, 20–22 May 2010; pp. 94–105.
30. Abeliuk, A.; Navarro, G. Compressed Suffix Trees for Repetitive Texts. In Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE), Cartagena, Colombia, 21–25 October 2012; pp. 30–41.
31. Navarro, G.; Mäkinen, V. Compressed full-text indexes. *ACM Comput. Surv.* **2007**, *39*, doi:10.1145/1216370.1216372.
32. Ferragina, P.; González, R.; Navarro, G.; Venturini, R. Compressed text indexes: From theory to practice. *ACM J. Exp. Algorithmics* **2009**, *13*, doi:10.1145/1412228.1455268.
33. Munro, I. Tables. In Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Hyderabad, India, 18–20 December 1996; pp. 37–42.
34. Larsson, J.; Moffat, A. Off-Line dictionary-based compression. *Proc. IEEE* **2000**, *88*, 1722–1732.
35. González, R.; Navarro, G. Compressed Text Indexes with Fast Locate. In Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM), London, Canada, 9–11 July 2007; pp. 216–227.
36. Mäkinen, V.; Navarro, G. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.* **2005**, *12*, 40–66.

37. Raman, R.; Raman, V.; Rao, S. Succinct Indexable Dictionaries with Applications to Encoding k-Ary Trees and Multisets. In Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), San Francisco, CA, USA, 6–8 January 2002; pp. 233–242.
38. Okanohara, D.; Sadakane, K. Practical Entropy-Compressed Rank/Select Dictionary. In Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, LA, USA, 6 January 2007.
39. González, R.; Grabowski, S.; Mäkinen, V.; Navarro, G. Practical Implementation of Rank and Select Queries. In Proceedings of the posters 4th Workshop on Experimental Algorithms (WEA), Santorini Island, Greece, 10–13 May 2005; pp. 27–38.
40. Claude, F.; Navarro, G. Practical Rank/Select Queries over Arbitrary Sequences. In Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE), Melbourne Australia, 10–12 November 2008; pp. 176–187.
41. Puglisi, S.; Turpin, A. Space-Time Tradeoffs for Longest-Common-Prefix Array Computation. In Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC), Gold Coast, Australia, 15–17 December 2008; pp. 124–135.
42. Kärkkäinen, J.; Manzini, G.; Puglisi, S. Permuted Longest-Common-Prefix Array. In Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM), Lille, France, 22–24 June 2009; pp. 181–192.
43. Brisaboa, N.; Ladra, S.; Navarro, G. Directly Addressable Variable-length Codes. In Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE), Saariselk, Finland, 25–27 August 2009; pp. 122–130.
44. Pizza & Chili Corpus. Available online: http://pizzachili.dcc.uchile.cl (accessed on 20 May 2013).
45. Geary, R.; Rahman, N.; Raman, R.; Raman, V. A simple optimal representation for balanced parentheses. *Theory Comput. Sci.* **2006**, *368*, 231–246.
46. Fischer, J.; Heun, V. Space-Efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.* **2011**, *40*, 465–492.
47. Konow, R.; Navarro, G. Faster Compact Top-k Document Retrieval. In Proceedings of the 23rd Data Compression Conference (DCC), Snowbird, UT, USA, 20 March 2013; pp. 351–360.
48. Sakamoto, H. A fully linear-time approximation algorithm for grammar-based compression. *J. Discret. Algorithms* **2005**, *3*, 416–430.
49. Pizza & Chili Repetitive Corpus. Available online: http://pizzachili.dcc.uchile.cl/repcorpus (accessed on 20 May 2013).
50. Software Page of Gonzalo Navarro. Available online: http://www.dcc.uchile.cl/gnavarro/software (accessed on 20 May 2013).
51. Pizza & Chili Corpus. Available online: http://pizzachili.dcc.uchile.cl/cst (accessed on 20 May 2013).