Practical Compressed Suffix Trees

Rodrigo Cánovas and Gonzalo Navarro

Department of Computer Science, University of Chile. {rcanovas|gnavarro}@dcc.uchile.cl

Abstract. The suffix tree is an extremely important data structure for stringology, with a wealth of applications in bioinformatics. Classical implementations require much space, which renders them useless for large problems. Recent research has yielded two implementations offering widely different space-time tradeoffs. However, each of them has practicality problems regarding either space or time requirements. In this paper we implement a recent theoretical proposal and show it yields an extremely interesting structure that lies in between, offering both practical times and affordable space. The implementation of the theoretical proposal is by no means trivial and involves significant algorithm engineering.

1 Introduction

The suffix tree [18, 30] is arguably the most important data structure for string analysis. It has been said to have a myriad of virtues [2] and there are even books dedicated to its applications in areas like bioinformatics [12]. Many complex sequence analysis problems are solved through sophisticated traversals over the suffix tree, and thus a fully-functional suffix tree implementation supports a variety of *navigation operations*. These involve not only the classical tree navigation operations (parent, child) but also specific ones such as suffix links and lowest common ancestors.

One serious problem of suffix trees is that they take much space. A naive implementation can easily require 20 bytes per character, and a very optimized one reaches 10 bytes [14]. A way to reduce this space to about 4 bytes per character is to use a simplified structure called a suffix array [17], but it does not contain sufficient information to carry out all the complex tasks suffix trees are used for. Enhanced suffix arrays [1] extend suffix arrays so as to recover the full suffix tree functionality, raising the space to about 6 bytes per character in practice. Some other heuristic space-saving methods [20] achieve about the same.

To have an idea of how bad is this space, consider that, on DNA, each character could be encoded with 2 bits, whereas the alternatives we have considered require 32 to 160 bits per character. Using suffix trees on secondary memory makes them orders of magnitude slower as most traversals are non-local. This 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.)

Recent research on *compressed suffix trees (CSTs)* has made much progress in terms of reaching space requirements that approach not only the worst-case space of the sequence, but even its information content. All these can be thought of as a *compressed suffix array (CSA)* plus some extra information that encodes the tree topology and longest common prefix (LCP) information.

The first such proposal was by Sadakane [27]. It requires $6n$ bits on top of his CSA [26], which in turn 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). It has recently been implemented by Välimäki

et al. [29]. They achieve a few tens of microseconds per operation, but in practice the structure requires about 25–35 bits per symbol (close to a suffix array), and thus its applicability is limited.

The second proposal was by Russo et al. [25]. It requires only $o(n)$ bits on top of a CSA. By using an FM-index [6] 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, for sufficiently low $k \le \alpha \log_\sigma n$, 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). This structure was implemented by Russo and shown to achieve very little space, around 4–6 bits per symbol, which makes it extremely attractive when the sequence is large compared to the available main memory. On the other hand, the structure is much slower than Sadakane's. Each navigation operation takes the order of milliseconds, which starts to approach disk operation times.

Both existing implementations are unsatisfactory in either time or space (though certainly excell on the other aspect), and become very far extremes of a tradeoff: Either one has sufficient main memory to spend 30 bits per character, or one has to spend milliseconds per navigation operation.

In this paper we present a third implementation, which offers a relevant space/time tradeoff between these two extremes. One variant shows to be superior to the implementation of Sadakane's CST in *both* space and time: it uses 13–16 bits per symbol (i.e., half the space) and requires a few microseconds (i.e., several times faster) per operation. A second alternative works within 8–12 bits per symbol 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.

Our implementation is based on a third theoretical proposal, by Fischer et al. [8], which achieves $nH_k(2\max(1,\log(1/H_k))+1/\epsilon+O(1))+o(n\log\sigma)$ bits per symbol (for the same k as above and any constant $\epsilon > 0$) and navigation times of the form $O(\log^\epsilon n)$. The paper contains several theoretical structures and solutions, whose efficient implementation was far from trivial, and required significant algorithm engineering that completely changed the original proposal in some cases. We study both the original and our proposed variants, and come up with the two variants mentioned above.

2 Compressed Suffix Trees

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 $1 \le i < n$. Note that every substring of T is the prefix of a suffix, and that all suffixes starting with a given pattern P appear consecutively in A, hence a couple 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)* [21, 5], 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 $j+1 = A[\Psi(i)]$.

A *suffix tree* is a compact 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 children of a node are labeled by different characters. Leaves are formed when the prefix of the corresponding suffix is already unique. Here "compact" means that unary paths are converted into a single edge, labeled by the string formed by concatenating the involved character labels. If the children of each node are ordered

lexicographically by their string label, then the leaves of the suffix tree form the suffix array of T. Fig. 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.

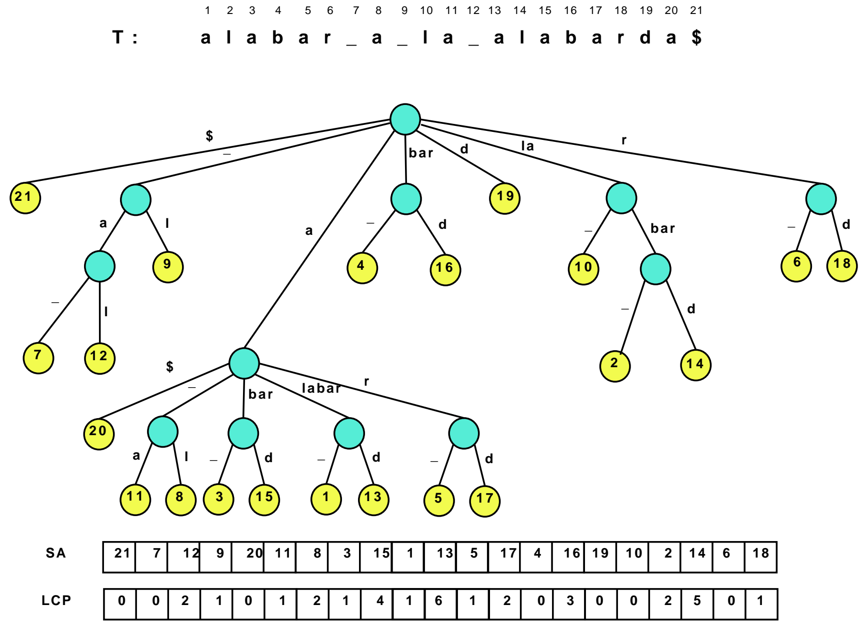

Fig. 1. The suffix tree of the text "alabar a la alabarda$", where the "$" is a terminator symbol. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters "a"-"z".

In order to get a suffix tree from a suffix array, one needs at most 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, the length of the string labeling the path from the root to the lowest common ancestor node of suffix tree leaves i and $i-1$). 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 [27] added the topology of the tree (using $4n$ extra bits) and the LCP data. The LCP 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 LCP' be the permuted LCP array, then $LCP'[j+1] \geq LCP'[j] - 1$. Thus the numbers can be differentially encoded, $h[j+1] = LCP'[j+1] - LCP'[j] + 1 \geq 0$, 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 $LCP'[A[i]]$, 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 [19]. Then $LCP'[j] = select_1(H, j) - 2j$, assuming $LCP'[0] = 0$.

Operation	Description
Root()	the root of the suffix tree.
Locate(v)	the suffix position i if v is the leaf of suffix $T_{i,n}$, otherwise NULL.
Ancestor(v, w)	true if v is an ancestor of w.
SDepth(v)/TDepth(v)	the string-depth/tree-depth of v.
Count(v)	the number of leaves in the subtree rooted at v.
Parent(v)	the parent node of v.
FChild(v)	the alphabetically first child of v.
NSibling(v)	the alphabetically next sibling of v.
SLink(v)	the 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)	the iterated suffix-link of v; i.e., the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma^i$.
LCA(v, w)	the lowest common ancestor of v and w.
Child(v, a)	the node w s.th. the first letter on edge (v, w) is $a \in \Sigma$.
Letter(v, i)	the *ith* letter of v's path-label, $\pi(v)[i]$.
LAQ$_S$(v, d)/LAQ$_T$(v, d)	the hightest ancestor of v with string-depth/tree-depth $\leq d$.

Table 1. Operations over the nodes and leaves of the suffix tree.

Russo et al. [25] 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 [27].

A New Theoretical CST Proposal. Fischer et al. [8] prove that array H in Sadakane's CST is indeed compressible as it has at most $2r \leq 2(nH_k + \sigma^k)$ *runs* of 0s or 1s, for any k. Let $z_1, z_2, \ldots, z_r$ the lengths of the runs of 0s and $o_1, o_2, \ldots, o_r$ the lengths of the runs of 1s. They create arrays $Z = 10^{z_1-1}10^{z_2-1}\ldots$ and $O = 10^{o_1-1}10^{o_2-1}\ldots$, which have overall $2r$ 1s out of $2n$, and hence can be compressed to $2r\log\frac{n}{r} + O(r) + o(n)$ bits while supporting constant-time *rank* and *select* [24].

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 under this representation by means of three operations: (1) $RMQ(i, j)$ gives the position of the minimum in $LCP[i, j]$; (2) $PSV(i)$ finds the last value smaller than $LCP[i]$ in $LCP[1, i-1]$; and (3) $NSV(i)$ finds the first value smaller than $LCP[i]$ in $LCP[i+1, n]$. All these could easily be solved in constant time using $O(n)$ extra bits of space on top of the LCP representation, but Fischer et al. give sublogarithmic-time algorithms to solve them with only $o(n)$ extra bits.

As examples, the parent of node $[i, j]$ is $[PSV(i), NSV(i) - 1]$; 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 $[PSV(\Psi(i)), NSV(\Psi(j)) - 1]$.

Our Contribution. The challenge faced in this paper is to implement this CST. This can be divided into (1) how to represent LCP efficiently in practice, and (2) how to compute efficiently RMQ, PSV, and NSV over this LCP representation. We study each subproblem separately and then compare the resulting CST with previous ones.

All our experiments were performed on 100 MB of the protein, sources, XML and DNA texts from Pizza&Chili corpus (*http://pizzachili.dcc.uchile.cl*). The computer used features an Intel(R) Core(TM)2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running version 2.6.24-24 Linux kernel.

3 Representing array LCP

The following alternatives were considered to represent LCP:

Sad-Gon Encodes H in plain, using the *rank*/*select* implementation of González [10], which takes $0.1n$ bits over the $2n$ used by H itself and answers *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 [22] for H. This requires about the same space as the previous one and answers *select* in $O(\log^4 r/\log n)$ time.

FMN-RRR Encoding H in compressed form as suggested by Fischer et al. [8], that is, by encoding bitmaps Z and O. We use the compressed representation by Raman et al. [24] as implemented by Claude [4]. This poses $0.54n$ bits of overhead on top of the entropy of the two bitmaps, $2r\log\frac{n}{r}+O(r)$. Operation *select* takes $O(\log n)$ time.

FMN-OS Like the previous one, but instead of Raman et al. technique, we use the *sparse array* implementation by Okanohara and Sadakane [22]. This requires $2r\log\frac{n}{r}+O(r)$ bits and solves *select* in time $O(\log^4 r/\log m)$.

PT Inspired on an LCP construction algorithm by Puglisi and Turpin [23], 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 relatively slow.

PhiSpare In the same spirit of the previous one, this is inspired in a construction by Kärkkäinen et al.[13]. For a parameter q, store in text order an array LCP'_q with the LCP values for all text positions $q\cdot k$. Now assume $SA[i]=qk+b$, with $0\leq b<k$. If $b=0$, then $LCP[i]=LCP'_q[k]$. Otherwise, $LCP[i]$ is computed by comparing at most $q+LCP'_q[k+1]-LCP'_q[k]$ symbols of the suffixes $T[SA[i-1],n]$ and $T[SA[i],n]$. 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. [3]. Most LCP values are small ($O(\log_\sigma n)$ on average), and thus one could use few bits to represent them. Yet, some can be much longer. 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, and finding the values of b that minimize the total space (*DAC-Var*). Note we represent LCP and not LCP', thus we do not need to compute $A[i]$.

RP *Re-Pair* [15] is a grammar-based compression method that factors out repetitions in a sequence. It has been used [11] to compress the differentially encoded suffix array, $SA'[i]=SA[i]-SA[i-1]$, which contains repetitions because SA can be partitioned into r areas that appear elsewhere in SA with the values shifted by 1 [16]. Note that LCP must then contain the same repetitions shifted by 1, and therefore Re-Pair compression of the differential LCP should perform similarly, as advocated in the theoretical proposal [8]. To obtain $LCP[i]$ we must store some sampled absolute LCP values and decompress the nonterminals since the last sample.

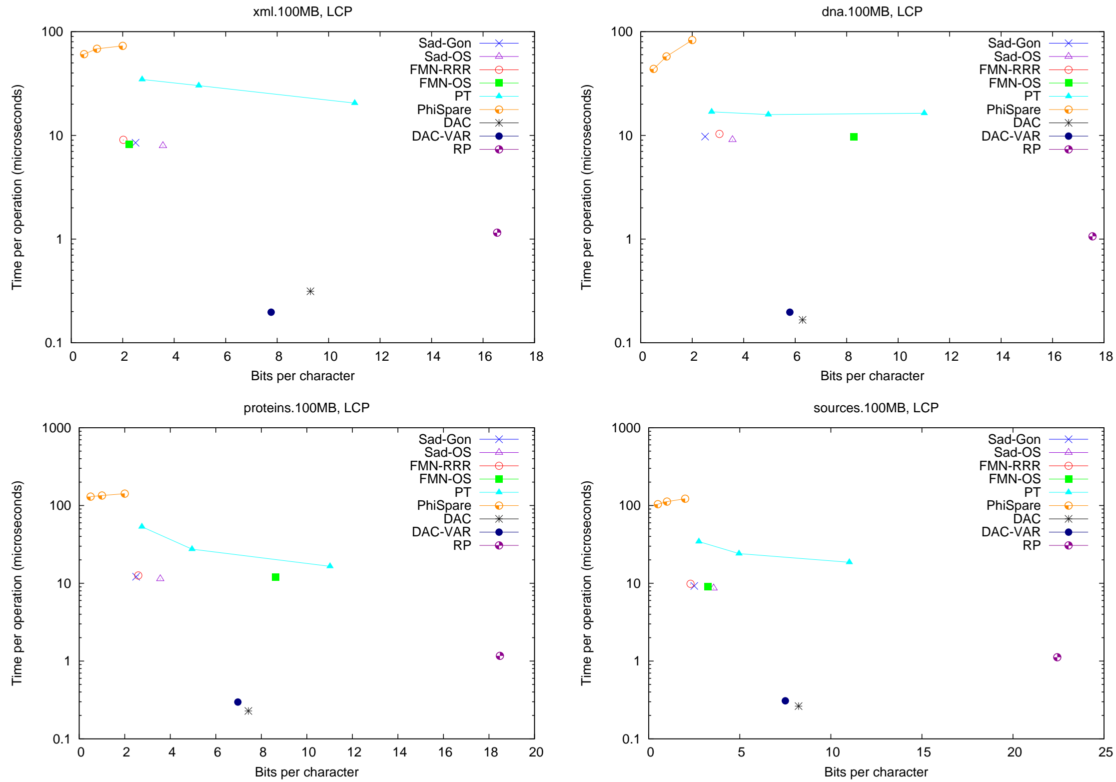

Fig. 2. Space/time for accessing LCP array.

Experimental Comparison. We tested the different LCP implementations by accessing 100,000 random positions of the LCP array. Fig. 2 shows the space/times achieved. Only *PT* and *PhiSpare* display a space/time tradeoff; in the first we use $v = 4, 6, 8$ and for the second $q = 16, 32, 64$.

As it can be seen, $DAC/DAC\text{-}Var$ and the representations of H dominate the space-time tradeoff map (*PhiSpare* and *PT* can use less space but they become impractically slow). For the rest of the paper we will keep only DAC and $DAC\text{-}Var$, which give the best time performance, and $FMN\text{-}RRR$ 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. [8]. For NSV, the idea is akin to the recursive *findclose* solution for compressed trees [9]: 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 an existing implementation [7] to the LCP array, remembering that we do not have direct access to LCP but have to use any of the access methods we have developed for it. This accesses at most 5 cells of LCP, yet it requires $3.25n$ bits. In the actual theoretical proposal [8] this is reduced to $o(n)$ but many more accesses to LCP would be necessary; we did not implement that verbatim as it has little chances of being practical.

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

4.1 A Novel Practical Solution

We propose now a different solution, inspired in Sadakane and Navarro's succinct tree representation [28]. 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 needed 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 considering 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. Note that the heaviest part of the cost in practice is the $O(L)$ accesses to LCP cells at the lowest levels, since the minima in T_m are explicitly stored.

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 *NPR-CN*.

Experimental Comparison. We tested the performance of the different NPR implementations by performing 100,000 NSV and RMQ queries at different random positions in the LCP array. Fig. 3 shows the space/time achieved for each implementation. We used the slower *Sad-Gon* implementation for LCP to enhance the differences in time performance. We obtained space/time tradeoffs by using different block sizes $L=8,16,32$ (so the times for RMQ on *NPR-FMN* are

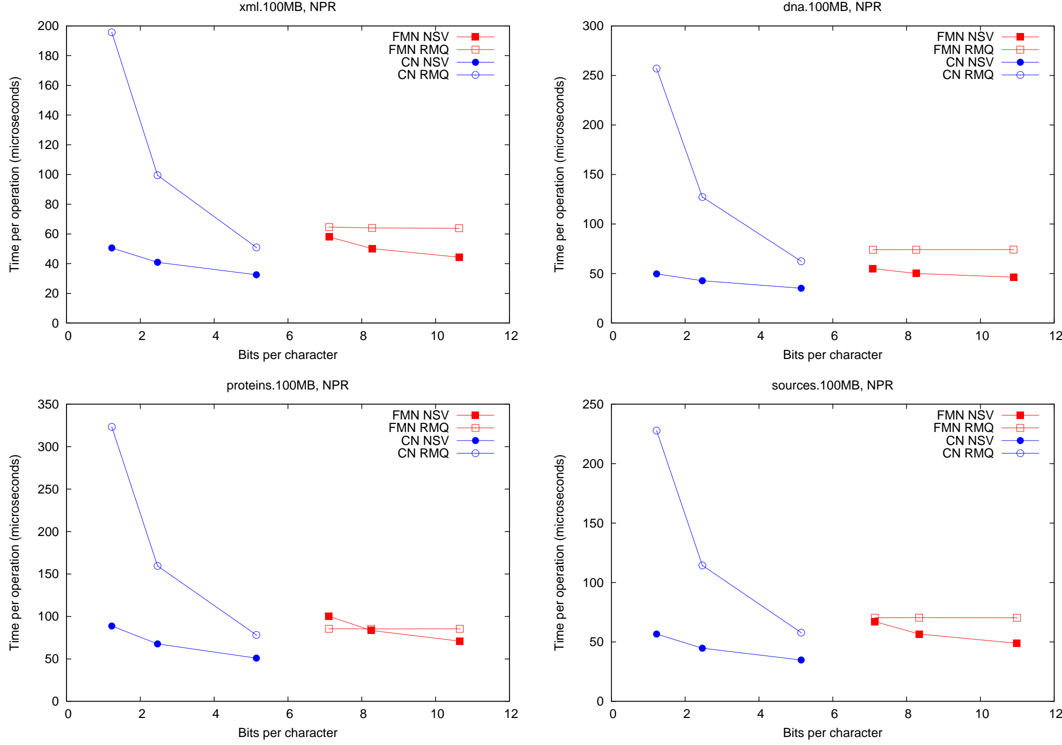

Fig. 3. Space/time for the operations NSV and RMQ.

not affected). Clearly NPR-CN displays the best performance for NSV, both in space and time. For RMQ, it can be see that the best time obtained with NPR-CN dominates, in time and space, the NPR-FMN curve. Thus NPR-CN is our chosen implementation for the rest.

5 Our Compressed Suffix Tree

Our CST implementation applies our NPR-CN 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. [8], which we do not repeat for lack of space. 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 proposal.

NSib: There is a bug in the original formula [8] in the case v is the next-to-last child of its parent. According to them, $NSib([v_l, v_r])$ 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 $z = NSV'(v_r + 1, LCP[v_r + 1])$, where $NSV'(i, d)$ is a generalization of NSV that finds 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 $NSib$, 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.

$\mathbf{LAQ}_S(v,d)$: Instead of the slow and 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.

$\mathbf{LAQ}_T(v,d)$: There is no practical solution in the original proposal. We proceed as follows to achieve the cost of d *Parent* operations, plus sume 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.

6 Comparing the CST Implementations

We compare all the CST implementations: Välimäki et al.'s [29] implementation of Sadakane's compressed suffix tree [27] *(CST-Sadakane)*; Russo's implementation of Russo et al.'s "fully-compressed" suffix tree [25] *(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 implementations; *Locate* and *Count* because they depend only on the underlying compressed suffix array (which is mostly orthogonal); $SLink^i$ 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.

We typically show space/time tradeoffs for all the structures, where the space is measured in bits per character of the original text (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 string depth larger than 5 (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). For space limitations, and because the outcomes are consistent across texts, we show the results of each operation over one text only, choosing in each case a different text.

Fig. 4 shows space/time tradeoffs for six operations. 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 bits per symbol (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 bits per symbol and carry out most operations within a few microseconds (the "large and fast" variants *DAC* and *DAC-Var*). In contrast, the FCST requires only 4–6 bits per symbol (which is, remarkably, up to half the space required by the plain text representation), but takes the order of milliseconds per operation; and Sadakane's CST takes usually a few tens of microseconds per operation but requires 25–35 bits per character, which is close to uncompressed suffix arrays (not uncompressed

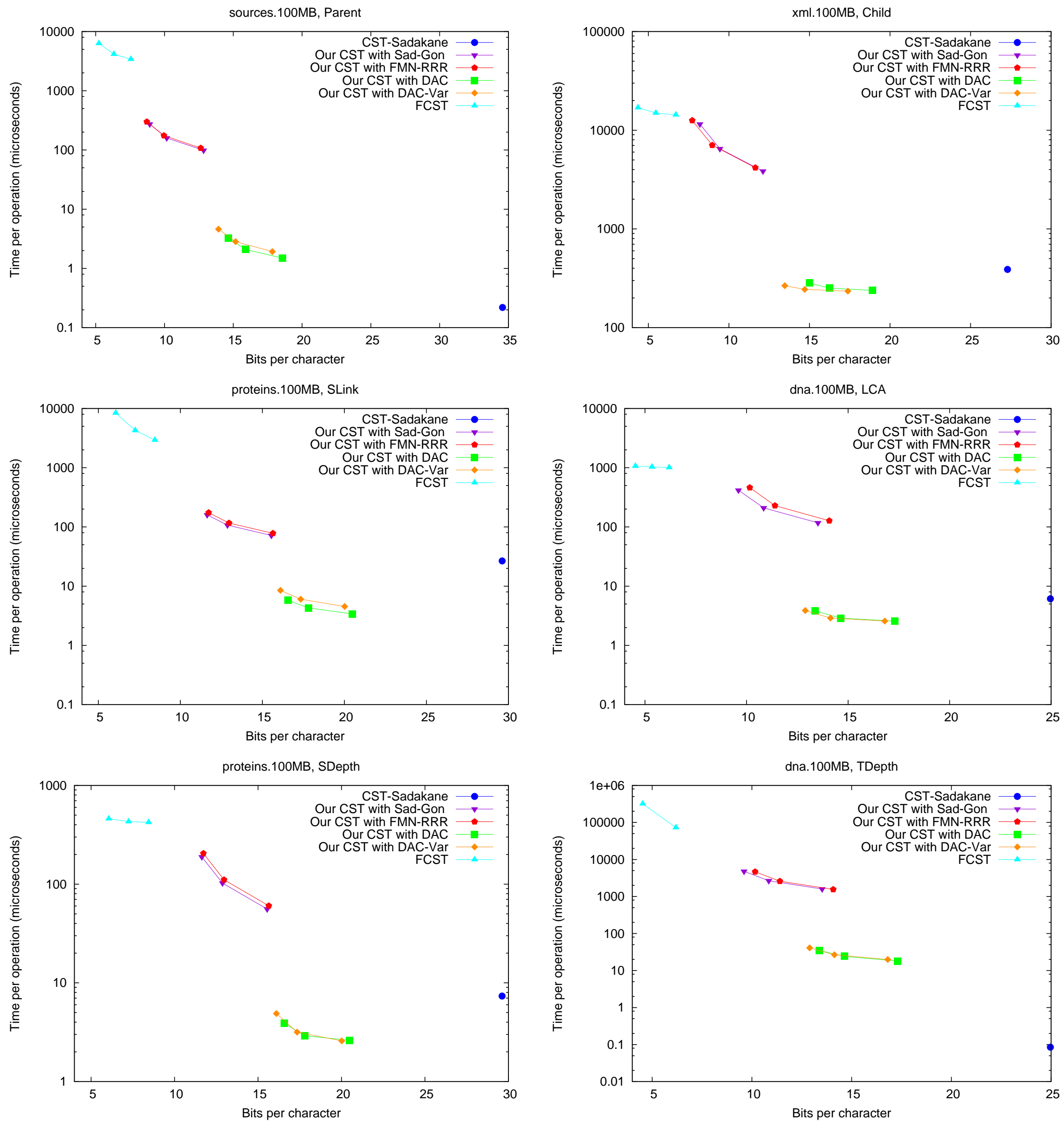

Fig. 4. Space/time tradeoff performance figures for different CSTs and suffix tree operations. Note the logscale.

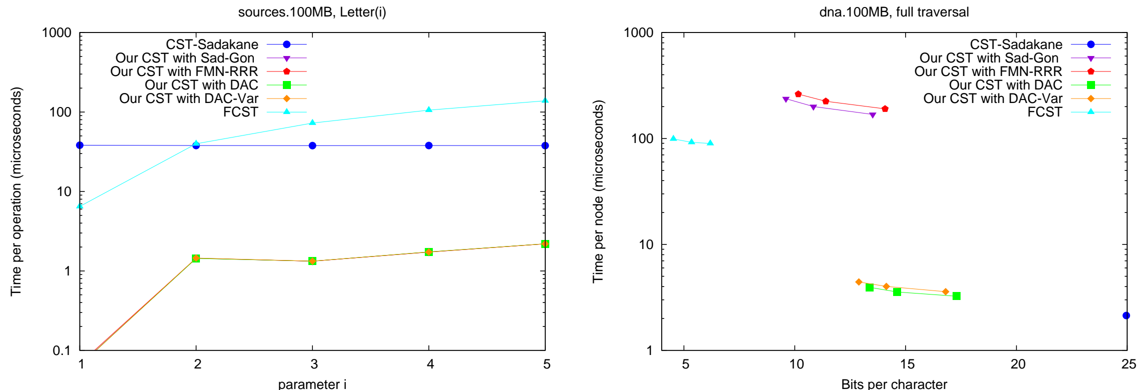

Fig. 5. Time comparison for operation *Letter* (left) and a full traversal computing *SDepth*. Note the logscale.

suffix trees, though). We remark that, for many operations, our "fast and large" variant still takes half the space of Sadakane's CST implementation and operates several times faster.

Exceptions to the latter comment are *Parent* and *TDepth*, where Sadakane's CST stores the explicit tree topology, and thus takes a fraction of a microsecond. On the other hand, our CST carries out LAQ_S (not shown) 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 the 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 $LAQ_T(v, d)$ (not shown) would cost about d times the time of our *TDepth*.

Fig. 5 (left) shows operation *Letter(i)* as a function of i. This requires either applying $i-1$ times Ψ, or applying once SA and SA^{-1}. The former choice is preferred for the FCST and the latter in Sadakane's CST. For our CST, using Ψ iteratively was better for these i values, as the alternative requires around 70 microseconds. Note this operation depends only on the CSA structure.

We finish with 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 *NSib* and reports the maximum *SDepth*. Fig. 5 (right) illustrates the result. Although Sadakane's CST takes advantage of locality, our "large and fast" variant is pretty close using half the space. Our "small and slow" variant, instead, requires a few hundred microseconds as expected, yet the FCST has a special implementation for full traversals and, this time, it beats our slow variant in space and time.

7 Final Remarks

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 plan to leave the code publicly available to foster its widest dissemination. We also plan to apply it to solve concrete bioinformatic problems on large instances.

Acknowledgements. Supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile. We thank F. Claude, J. Fischer, R. González, J. Kärkkäinen, S. Ladra, V. Mäkinen, S. Puglisi, L. Russo, and K. Sadakane for code, help to use it, and discussions.

References

1. M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discr. Algorithms*, 2(1):53–86, 2004.
2. A. Apostolico. *The myriad virtues of subword trees*, pages 85–96. Combinatorial Algorithms on Words. NATO ISI Series. Springer-Verlag, 1985.
3. N. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. 16th SPIRE*, LNCS 5721, pages 122–130, 2009.
4. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.
5. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Exp. Algor.*, 13:article 12, 2009.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algor.*, 3(2):article 20, 2007.
7. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. 1st ESCAPE*, LNCS 4614, pages 459–470, 2007.
8. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comp. Sci.*, 410(51):5354–5364, 2009.
9. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comp. Sci.*, 368:231–246, 2006.
10. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th WEA (posters)*, pages 27–38, 2005.
11. R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th CPM*, LNCS 4580, pages 216–227, 2007.
12. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
13. J. Karkkainen, G. Manzini, and S. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th CPM*, LNCS 5577, pages 181–192, 2009.
14. S. Kurtz. Reducing the space requirements of suffix trees. *Soft. Pract. Exp.*, 29(13):1149–1171, 1999.
15. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.
16. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. Comp.*, 12(1):40–66, 2005.
17. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, pages 935–948, 1993.
18. E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 32(2):262–272, 1976.
19. I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
20. I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *J. Algor.*, 39(2):205–222, 2001.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
22. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, 2007.
23. S. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. 19th ISAAC*, pages 124–135, 2008.
24. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
25. L. Russo, G. Navarro, and A. Oliveira. Fully-Compressed Suffix Trees. In *Proc. 8th LATIN*, LNCS 4957, pages 362–373, 2008.
26. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algor.*, 48(2):294–313, 2003.
27. K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.
28. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st SODA*, pages 134–149, 2010.
29. N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen. Engineering a compressed suffix tree implementation. In *Proc. 6th WEA*, pages 217–228, 2007.
30. P. Weiner. Linear pattern matching algorithms. In *IEEE Symp. Swit. and Aut. Theo.*, pages 1–11, 1973.