

Succinct Trees in Practice

Diego Arroyuelo*

Rodrigo Cánovas†

Gonzalo Navarro†

Kunihiko Sadakane‡

Abstract

We implement and compare the major current techniques for representing general trees in succinct form. This is important because a general tree of n nodes is usually represented in pointer form, requiring $O(n \log n)$ bits, whereas the succinct representations we study require just $2n + o(n)$ bits and carry out many sophisticated operations in constant time. Yet, there is no exhaustive study in the literature comparing the practical magnitudes of the $o(n)$ -space and the $O(1)$ -time terms. The techniques can be classified into three broad trends: those based on BP (balanced parentheses in preorder), those based on DFUDS (depth-first unary degree sequence), and those based on LOUDS (level-ordered unary degree sequence). BP and DFUDS require a balanced parentheses representation that supports the core operations *findopen*, *findclose*, and *enclose*, for which we implement and compare three major algorithmic proposals. All the tree representations require also core operations *rank* and *select* on bitmaps, which are already well studied in the literature. We show how to predict the time and space performance of most variants via combining these core operations, and also study some tree operations for which specialized implementations exist. This is especially relevant for a recent proposal (K. Sadakane and G. Navarro, *SODA'10*) which, although belonging to class BP, deviates from the main techniques in some cases in order to achieve constant time for the widest range of operations. We experiment over various types of real-life trees and of traversals, and conclude that the latter technique stands out as an excellent practical combination of space occupancy, time performance, and functionality, whereas others, particularly LOUDS, are still interesting in some limited-functionality niches.

1 Introduction.

Trees are, on one hand, the paradigmatic data structure in Computer Science, probably rivalled in popularity only by arrays and linked lists; and on the other hand, one of the most striking examples of the success of succinct data structures. A classical representation of a general tree of n nodes requires $O(nw)$ bits of space, where $w \geq \log_2 n$ is the bit length of a machine pointer (we will use \log for \log_2 henceforth). The associated constant is at least 2, and typically only operations such as moving to the first child and to the next sibling, or

to the i -th child, are supported. By further increasing the constant, some other simple operations are easily supported, such as moving to the parent, knowing the subtree size, or the depth. Much research was needed (and further increase of the constant) to support sophisticated operations such as level-ancestor [3] and lowest common ancestor [2] queries. While the constant-time complexities achieved for the sophisticated queries are remarkable, the $\Omega(n \log n)$ -bit space complexity is not justified in terms of Information Theory: there are only $C_n \sim 4^n/n^{3/2}$ different general trees of n nodes, and thus $\log C_n = 2n - \Theta(\log n)$ bits are sufficient to distinguish any one of them. This huge space gap has motivated a large body of research [16, 19, 20, 21, 11, 4, 10, 7, 15, 12, 25, 17, 18, 8, 27] achieving $2n + o(n)$ bits of space and constant time for an impressive set of operations. Table 1 lists those we consider in this paper. There are several others, yet most are solved analogously to those in this set. Note we consider also *labeled* trees, where each child of a node is associated to a distinct label in a range $[1, \sigma]$. In this case, $n \log \sigma$ additional bits are needed to represent the labels.

The existing proposals differ in their functionality, ranging from those that support basically child/parent navigation [16, 7] to those supporting a full range of operations [4, 17, 8, 27]; in the nature of the $o(n)$ space overhead, ranging from $O(n/(\log \log n)^2)$ [18] to $O(n/\text{polylog}(n))$ [27]; and in some rare cases on their capability to take less space when the tree is compressible in some sense [17]. In theoretical terms, the problem can be considered basically solved, at least in the static scenario.

In practice, however, the situation is much less satisfactory. Implementations of the theoretical proposals are scarce and far from trivial, even for those striving for simplicity [10, 7, 27]. Verbatim implementations of the theoretical proposals are usually disastrous, as already noted in previous work [22], and much empirical experience must be combined with the theoretical ideas in order to obtain theoretically and practically sound solutions. There are few practical comparisons among techniques, none of which is sufficiently exhaustive to give a broad idea of their performance. In practice, the $O(1)$ -time and the $o(n)$ -space terms in the asymptotic analyses give little clues as to how the structures actu-

*Yahoo! Research Latin America, Chile. darroyue@dcc.uchile.cl.

†Department of Computer Science, University of Chile. [rcanovas|gnavarro}@dcc.uchile.cl](mailto:{rcanovas|gnavarro}@dcc.uchile.cl). Supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

‡National Institute of Informatics (NII), Japan. sada@nii.ac.jp. Supported in part by a Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

| parentheses operation | description |
|----------------------------|---|
| $findclose(i)/findopen(i)$ | position of parenthesis matching $P[i]$ |
| $enclose(i)$ | position of tightest opening parenthesis enclosing node i |
| $rank_c(i)/rank_c(i)$ | number of opening/closing parentheses in $P[1, i]$ |
| $select_c(i)/select_c(i)$ | position of i -th open/close parenthesis |
| tree operation | description |
| $pre_rank(x)$ | preorder rank of node x |
| $pre_select(x)$ | the node with preorder x |
| $isleaf(x)$ | whether node x is a leaf |
| $ancestor(x, y)$ | whether x is an ancestor of y |
| $depth(x)$ | depth of node x |
| $parent(x)$ | parent of node x |
| $first_child(x)$ | first child of node x |
| $next_sibling(x)$ | next sibling of node x |
| $subtree_size(x)$ | number of nodes in the subtree of node x |
| $degree(x)$ | number of children of node x |
| $child(x, i)$ | i -th child of node x |
| $child_rank(x)$ | number of siblings to the left of node x |
| $level_ancestor(x, d)$ | ancestor y of x such that $depth(y) = depth(x) - d$ |
| $lca(x, y)$ | the lowest common ancestor of two nodes x, y |
| labeled tree operation | description |
| $labeled_child(x, s)$ | child of node x labeled by symbol s |
| $child_label(x)$ | label of edge between node x and its parent |

Table 1: Operations on parentheses and trees considered in this paper. The first group are core operations on a parentheses sequence P , and is used to implement many of the others.

ally perform.

In this paper, we carefully implement the theoretical proposals we consider most promising in practice, and compare them for various real-life trees and typical tree traversals. Tree representations can be broadly classified into the following tracks:

BP: The *balanced parentheses* representation of a tree, first advocated as a succinct data structure by Jacobson [16], and later achieving constant times [19], is built from a depth-first preorder traversal of the tree, writing an *opening* parenthesis when arriving to a node for the first time, and a *closing* parenthesis when going up (after traversing the subtree of the node). In this way, we get a sequence of $2n$ balanced parentheses. Each node is represented by a matching pair parentheses '(' and ')', and we identify the node with its opening parenthesis. The subtree of x contains those nodes (parentheses) enclosed between its representing opening and closing parentheses. The core operations (see Table 1) on this sequence are sufficient to implement most of the functionality.

DFUDS: Depth-first unary degree sequence [4]. It is built by the same traversal, except that, upon arriving at each node, we append i opening and one

closing parentheses, being i the number of children of the node. The node is represented by the position where its i parentheses start. The resulting sequence turns out to be balanced (if one prepends an extra opening parenthesis at the beginning) and the same core operations on parentheses are used to support the same functionality of BP in a different way (except for *depth*, which requires extra structures [17]) plus others, most notably *child*, in constant time.

LOUDS: Level-ordered unary degree sequence [16, 4]. Nodes are processed and represented as in DFUDS, but they are traversed level-wise, instead of in depth-first order. It turns out that just *rank_c/rank_c*, and *select_c/select_c*, are sufficient to support a few key operations such as *parent* and *child* in constant time, yet most other operations are not supported.

FF: The recent so-called fully-functional succinct tree representation [27] is based on a BP representation. It includes a novel data structure to handle the core operations, called a *range min-max tree*. With little extra data, this same structure is able of solving in constant time many other sophisticated operations that are not usually handled by other BP represen-

tations, such as *child*, *lca*, and *levelAncestor*.

We omit an extra track based on tree covering [11, 15, 8], of which we do not know of practical implementations (we thank Arash Farzan and Meng He for confirming this).

We first study different alternatives to represent a sequence of balanced parentheses while supporting the core operations. These include a hash-based heuristic implementation developed for a compressed text index [22], Geary et al.’s recursive pioneer-based representation [10], and the range min-max trees [27]. We exhaustively compare these operations over real-life trees coming from (a) LZ78 parsings of text collections, (b) suffix trees of text collections, (c) the tag structure of XML collections. We emulate typical root-to-leaf (and vice versa) traversals coming from exact search, LZ78 decompression, and other very common navigation operations, as well as traversals where in addition every other child of the current node is visited with a given probability. These are meant to emulate traversals typical of backtracking, range or proximity searches, and XPath queries.

We briefly survey the way BP, DFUDS, and LOUDS build on these core operations, and show how to predict the performance of most operations in terms of the core ones. We also study some tree operations individually, when the prediction from the performance of core operations is less direct. Overall, our work provides implementations (all by ourselves) of the most promising techniques to represent succinct trees, the first exhaustive empirical comparison among them, and recommendations on which structures to use depending on the case. It turns out that the recent FF proposal [27] stands out as an excellent combination of wide functionality, little space usage, and good time performance. Other approaches are still relevant for certain niches. In particular, we show that LOUDS, which had received comparatively little attention, is an extremely simple and efficient alternative when only the simpler operations are needed. LOUDS excels when descending to an arbitrary ordinal or labeled child. For this latter operation, the DFUDS variant also shows to be a reasonable compromise if a wide functionality must be retained.

2 Balanced Parentheses.

Let $P[1, 2n]$ be a sequence of n pairs of balanced parentheses (represented with bits 0,1). We describe the three major representations we will consider, which support the core parenthesis operations described in Table 1.

Let us define the key function $excess(i) = rank_c(i) - rank_b(i)$. Then it holds that $findclose(i)$ is the smallest

$j > i$ such that $excess(j) = excess(i - 1)$ ($findopen(i)$ is analogous). Also, $enclose(i)$ is the greatest $j < i$ such that $excess(j) = excess(i) - 1$.

2.1 Hash-Based Heuristic (HB).

Instead of storing all the possible answers to $findclose$ (which would require $O(n \log n)$ bits of space), this representation [22] divides the parentheses into three groups. Let $b = \log n$ and $s = b \log n$. Let *close* parentheses be those whose matching parentheses are at distance at most b ; *near*, between $b + 1$ and s ; and *far*, farther than s positions away. Only the answers for near and far parentheses are explicitly stored, in hash tables. The table for far parentheses uses $O(\log n)$ bits per value stored, while that for near parentheses uses just $\log s$ bits (because the distance from the argument is stored, rather than the absolute answer).

To compute $findclose(i)$, we first scan the next b positions, looking for the first parenthesis with the same $excess(i - 1)$. If the answer is not found in this way, the hash table for near parentheses is queried. Since the search key is not stored in this table, we consider all colliding answers: if more than one candidate answer is found with the correct excess, the closest one is the correct answer (because the sequence is balanced [22]). Finally, if the answer is still not found, the hash table for far parentheses is queried. A similar approach is used for operations $findopen$ and $enclose$, requiring their own hash tables.

The search for close parentheses is implemented as a scan by chunks of k bits. Hence, for every different k -bit stream, we precompute its excess and, for every $j = 1, \dots, k$, the first position of the k -bit stream where the excess becomes $-j$, if any. To solve $findclose(i)$, we check whether the excess -1 is reached in the first chunk after position i . If it is, the position where this happens is $findclose(i)$. Otherwise, we consider the second chunk, looking for excess $-1 - e_1$, where e_1 is the total excess of the first chunk, and so on. If the b bits are exhausted without a solution, then the answer is not close. Scanning for $findopen(i)$ and $enclose(i)$ is analogous.

The parenthesis operations are thus supported in $O(b/k)$ time, plus two searches on hash tables, which are $O(1)$ on average. If we set $k = \frac{\log n}{c}$, for constant $c > 1$, the average time is $O(c)$. The precomputed table for close parentheses requires $2^k(2k + 1) \log k = O(n^{1/c} \log n \log \log n) = o(n)$ bits. Furthermore, we need $\log s = O(\log \log n)$ bits per near parenthesis and $O(\log n)$ per far parenthesis. However, there is no theoretical bounds on these types of parentheses (e.g., on a tree formed by n nodes with one child, almost all of them are far). Fortunately, these worst-case sequences

are not common in practice.

Implementation notes. We choose $k = 8$ so as to handle bytes, hence the table requires just 4.25 KB and admits good caching. The hash tables are implemented by a closed hashing scheme, with load factor 1.8 for *enclose* and 1.6 for *findclose* and *findopen*, which gave us the best space/time tradeoff (recall that we do not store the keys in the hash table for near parentheses, so we pay a noticeable cost per collision, and the cost is proportional to that of an unsuccessful search). See [22, Sect. 6] for more implementation details.

For operation *excess*, a lightweight *rank/select* data structure is used [13], which supports *rank* in $O(1)$ and *select* in $O(\log n)$ time, while requiring 5% extra space over that of $P[1, 2n]$. The idea is to store *rank* values, in 32 bits, every $s = 20 \cdot 32 = 640$ bits, and then scan byte-wise these 640 bits = 80 bytes using a precomputed table. For *select* we binary search the sampled values and finish with a sequential scan as well.

2.2 Recursive Pioneer-Based Representation (RP).

The representation by Geary et al. [10] divides P into blocks of size $b = \frac{\log n}{2}$. A parenthesis at position i is said to be *close* if its matching pair belongs to the same block, otherwise the pair is called *far*. Computing *findclose* for a close parenthesis is done by using precomputed tables, as in Section 2.1, requiring $o(n)$ bits of space. An opening far parenthesis at position i is said to be a *pioneer* if the previous opening far parenthesis has its matching closing parenthesis in a block different to that of the closing parenthesis for i . Both parentheses forming the matching pair are called pioneers. The total number of pioneers is $O(n/b)$ [16, 10]. Thanks to the pioneer properties, *findclose*(i) is reduced to (a) finding the previous pioneer $i' \leq i$ (which could be i itself), (b) find $j' = \text{findclose}(i')$, (c) scan the block of j' backwards until finding $j = \text{findclose}(i)$ using the *excess* property (again by means of precomputed tables).

To solve *findclose* for pioneers, we form the *reduced sequence* of the pioneers, and apply the solution recursively once more. The second time the reduced sequence is of length $O(n/b^2)$ and thus we can store all the answers in $O(n \log(n)/b) = o(n)$ bits.

The remaining piece is a bitmap $B[1, 2n]$ marking the pioneers. With rank_1 and select_1 on this sequence we find the previous pioneer, map it to the reduced sequence, and map the answer to *findclose* back to the original sequence. As B has $O(n/b)$ 1s, it can be represented using $O(n \log b/b) = o(n)$ bits of space [24].

The solution to *enclose*(i) is similar. After checking within the same block of i , we look for the first pioneer

c following i . If it is a closing parenthesis, then $p = \text{findopen}(c)$, otherwise p is the pioneer most tightly enclosing c (i.e., its *parent* in the sequence of pioneers). Now let q be the first pioneer following p . If q is in the same block as p , then our answer is the first far parenthesis to the left of q ; else the answer is the rightmost far parenthesis in the block of p . Both require just within-block scans. See the original article [10] for more details.

Implementation notes. For compressed bitmaps [24] we use a recent implementation [6] which requires 27% extra space on top of the entropy of the bitmap, and implements *rank* in constant time and *select* in $O(\log n)$ time. The block size b is a space/time tradeoff parameter. Because the second-level bitmap B is significantly smaller and not too compressible, we represent it in uncompressed form. All uncompressed RP bitmaps are implemented as those in Section 2.1.

2.3 Range Min-Max-Tree Based Representation (RMM).

The basic tool of this technique [27] is the *range min-max tree*. This is built over the (virtual) array of *excess*(i) values. The sequence P is cut into blocks of $b = \frac{w}{2}$ parentheses, and we store the minimum and maximum (local) excess within each block. We then group k blocks into a *superblock*, and store the minimum and maximum excess per superblock. We then group k superblocks, and so on until building up a complete k -ary hierarchy. The total space is $O(n \log(b)/b) = o(n)$ bits. The range min-max tree permits solving the following kernel operations. Let us overload $\text{excess}(i, j) = \text{excess}(j) - \text{excess}(i - 1)$.

fwd_search(i, d): gives the minimum $j > i$ such that $\text{excess}(i, j) = d$

bwd_search(i, d): gives the maximum $j < i$ such that $\text{excess}(j, i) = d$

With the kernel operations, we easily implement the basic parentheses operations, and also some sophisticated tree operations:

$$\begin{aligned} \text{findclose}(x) &\equiv \text{fwd_search}(x, 0) \\ \text{findopen}(x) &\equiv \text{bwd_search}(x, 0) \\ \text{enclose}(x) &\equiv \text{bwd_search}(x, 2) \\ \text{level_ancestor}(x, d) &\equiv \text{bwd_search}(x, d + 1) \end{aligned}$$

To solve *fwd_search*(i, d) we first scan the block of i in constant time using precomputed tables as before. If unsuccessful, we climb up the range min-max tree until we find a min/max excess range that, translated into absolute, contains $\text{excess}(i - 1) + d$: At each node, we scan the values to the right of the ancestor of i . Once the proper range min-max tree node is found, we go

down finding the first child that contains the desired excess, until reaching the leaf block, which is scanned to find the exact j value. The process is analogous for *bwd_search*.

If $n = \text{polylog}(w)$ and $k = \Theta(w/\log w)$, then there are $O(1)$ levels and the scanning per node can be done in constant time using universal tables of size $o(2^w)$. A constant-time solution for large trees using $2n + O(n/\text{polylog}(n))$ bits is significantly more complex [27], so we use range min-max trees for all cases. As a result, the complexity becomes $O(\log n)$ for all operations, yet the structure is extremely efficient anyway.

Implementation notes. For each block i , we store its local minimum $m[i]$ and maximum $M[i]$ excess values. Thus $-b \leq m[i] \leq 1$ and $-1 \leq M[i] \leq b$ holds and they can be stored in $\lceil \log(b+2) \rceil$ bits. Therefore the maximum value of b is 254 ($2^{16} - 2$) if we use one byte (two bytes) to store each value. For example, with $b = 512$ the space overhead is $2 \times 16 \cdot (2n)/b = 6.25\%$.

A superblock consists of k blocks, and we build a range min-max tree on superblocks, yet thereafter we use a branching factor 2, forming a complete binary tree. Storing 32-bit numbers and using, say, $k = 32$ and $b = 512$, the space is $2 \times 2 \cdot 32 \cdot (2n)/(bk) \approx 0.78\%$ overhead.

Just as for the other solutions, sequence P is scanned by bytes when looking for some target *excess* value. Local excesses are converted into global by adding *excess* at the beginning of blocks/superblocks. Since $\text{excess}(i) = \text{rank}_c(i) - \text{rank}_s(i) = 2 \cdot \text{rank}_c(i) - i$, all we need is a particularly efficient rank_c at the beginning of superblocks. Thus we adapt a *rank/select* scheme similar to the one used for the other representations [23], so that the *rank* structures use blocks aligned with b and the scheme requires little space. We store 4-byte global sums of big blocks of size 2^{16} , 2-byte local sums of blocks of size b , and 1-byte sums of blocks of 255 bits. For $b = 512$ this is $32 \cdot (2n)/2^{16} + 16 \cdot (2n)/b + 8 \cdot (2n)/255 \approx 6.31\%$ overhead.

3 Succinct Tree Representations.

3.1 Preorder Balanced Parentheses (BP).

Table 2 shows the way many operations in BP representation are expressed in terms of basic parenthesis operations (we assume the operations can be applied, otherwise an easy check is necessary). Operation $\text{child}(x, i)$ can be computed in $O(i)$ time by applying *first_child* and then, repeatedly, *next_sibling*. Operations $\text{degree}(x)$ and $\text{child_rank}(x)$ are solved similarly. Similarly, we implement $\text{level_ancestor}(x, d)$ in $O(d)$ time by successive *parent* operations. There are constant-time theoretical

solutions for these operations [21, 18], but they have not been implemented.

A complex operation that has a practical solution is $\text{lca}(x, y) = \text{enclose}(\text{rmq}(x, y) + 1)$ [25]. Operation $\text{rmq}(x, y)$ gives the minimum excess in $P[x, y]$, and is implemented in constant time using $O(n(\log \log n)^2 / \log n) = o(n)$ bits of space on top of the excess array [2, 25] (we simulate this excess array with operation *excess*).

Implementation notes. The most practical implementation we know of for *rmq* [9] requires 62.5% extra space on top of $P[1, 2n]$, and this would put the representation out of competition in terms of space requirement. A theoretical, not implemented, alternative solution to $\text{lca}(x, y)$ is given by operation *double-enclose* [19]. We opt, therefore, by taking *parent* on the less deep node until it becomes an *ancestor* of the deeper one, then this is the *lca*.

Labeled trees. The labels are stored in pre-order in a sequence $L[1, n]$, so that $\text{child_label}(x) = L[\text{pre_rank}(x)]$ and $\text{labeled_child}(x, s)$ is solved by a linear scan, just like $\text{child}(x, i)$.

3.2 Depth-First Unary Degree Sequence (DFUDS).

Table 2 also shows how to translate many operations in DFUDS representation to those on balanced parentheses. We have introduced shorthands $\text{prev}_t(x) \equiv \text{select}_t(\text{rank}_s(x))$ and $\text{next}_t(x) \equiv \text{select}_t(\text{rank}_s(x) + 1)$. If x is within a sequence of opening parentheses, these operations find the preceding and following closing parenthesis, respectively.

There is a theoretical proposal for implementing *depth* and *level_ancestor* in constant time [17], but this has not been implemented as far as we know. Again, $\text{lca}(x, y) = \text{parent}(\text{rmq}(x, y - 1) + 1)$ can be efficiently implemented in DFUDS [17], where *rmq* works over the excesses of the DFUDS parentheses sequence. (The formula needs an easy fix if $\text{ancestor}(x, y)$.)

Implementation notes. For the same practical reasons related to *rmq* and the non-implemented theoretical proposals, we implement *lca*, *level_ancestor*, and *depth* by brute force. For *lca* this means that we have to trace the full path from both nodes towards the root using *parent*, and then find the lowest common node.

We solve *prev*, and *next*, by directly scanning P computer-word-wise, which is preferable to the verbatim solution unless the tree has extremely large arities.

Labeled trees. The labels are written in a separate sequence $L[1, n]$ by traversing the tree in pre-order and writing, at each node, the symbols by which its children descend. In a labeled tree one can assume the children are ordered by their sym-

| Operation | BP | DFUDS | LOUDS |
|--------------------|------------------------------|---------------------------------------|--|
| $pre_rank(x)$ | $rank_c(x)$ | $rank_b(x-1) + 1$ | |
| $pre_select(p)$ | $select_c(p)$ | $select_b(p-1) + 1$ | |
| $isleaf(x)$ | $P[x+1] = ')$ | $P[x] = ')$ | $P[x] = ')$ |
| $ancestor(x, y)$ | $x \leq y \leq findclose(x)$ | $x \leq y \leq findclose(enclose(x))$ | |
| $depth(x)$ | $excess(x)$ | | |
| $parent(x)$ | $enclose(x)$ | $prev_b(findopen(x-1)) + 1$ | $prev_b(select_c(rank_b(x-1))) + 1$ |
| $first_child(x)$ | $x + 1$ | $child(x, 1)$ | $child(x, 1)$ |
| $next_sibling(x)$ | $findclose(x) + 1$ | $findclose(findopen(x-1)-1) + 1$ | $select_b(select_c(rank_b(x-1)) + 1) + 1$ |
| $subtree_size(x)$ | $(findclose(x) - x + 1) / 2$ | $(findclose(enclose(x)) - x) / 2 + 1$ | |
| $degree(x)$ | | $next_b(x) - x$ | $next_b(x) - x$ |
| $child(x, i)$ | | $findclose(next_b(x) - i) + 1$ | $select_c(rank_c(x) + i - 1) + 1$ |
| $child_rank(x)$ | | $next_b(y) - y; y = findopen(x-1)$ | $y - prev_b(y); y = select_c(rank_b(x-1))$ |

Table 2: Constant-time reduction of tree operations in BP and DFUDS, to operations on balanced parentheses.

bol. Thus to carry out $labeled_child(x, s)$ we binary search $L[rank_c(x), rank_c(x) + degree(x) - 1]$ and finish with a $child(x, i)$ operation. Also $child_label(x) = L[rank_c(parent(x)) + child_rank(x) - 1]$. By sorting the children labels of each parent in reverse order we save some $prev_/next_$ operations. There is a constant-time theoretical proposal for $labeled_child$ [4], not implemented as far as we know.

3.3 Level-Ordered Unary Degree Sequence (LOUDS).

The final column of Table 2 shows how some operations can be solved under LOUDS representation. We assume that, just as for DFUDS, a tree node corresponds to the first position of P where we write its arity in unary. The formulas differ slightly from the original version [16], but not significantly in terms of performance. Again, $first_child$ and $next_sibling$ can be expressed in terms of the others.

Other operations are not immediately supported. We discuss, however, some practical alternatives. To begin, $rank_b/$ $select_b$ provide a usually sufficient alternative to pre_rank/pre_select , as they map the tree nodes to consecutive numbers in $[1, n]$. In most cases the preorder primitives are used not because preorder is particularly relevant, but just for the sake of storing satellite information associated to the node identifiers in $[1, n]$. For this purpose, level-wise numbering is as good as preorder.

Although $ancestor(x, y)$ is not supported, there is a property of LOUDS that simplifies a brute-force implementation: *if y is deeper than x , then $y > x$* . Then, while $y > x$, we take $y \leftarrow parent(y)$. At the end, either $x = y$ (in which case the answer is yes) or $y < x$ (in which case the answer is no).

This same property permits a lightweight brute-

force implementation of $lca(x, y)$: If $y > x$ do $y \leftarrow parent(y)$; else if $x > y$ do $x \leftarrow parent(x)$. Continue until $x = y$, which is the answer.

Another operation that LOUDS lacks is $subtree_size(x)$. Since the nodes belonging to the subtree of x are contiguous at each level, this can be implemented by carrying an interval $[y, z]$ along the levels and counting the number of nodes at each level (with $rank_b(z) - rank_b(y-1) + 1$). Initially $[y, z] \leftarrow [x, x]$. For each next level we update $y \leftarrow child(y, 1)$ and $z \leftarrow child(z, degree(z))$, until $z < y$.

The other operations, $level_ancestor$ and $depth$, are implemented by brute force using $parent$ without any special improvements.

Implementation notes. We use the $rank/$ $select$ solution requiring 5% of extra space [13]. Several special cases are avoided by prepending $'()$ ' at the beginning of P . In a previous practical study on LOUDS [7] they choose differently the bit that represents a tree node. We found that they require more operations than us and than Jacobson [16] (except that they get $next_sibling$ almost for free) and their space overhead is too high (50%-100%).

Labeled trees. These are solved just as for DFUDS (reversing the order of children is not necessary).

3.4 A Fully-Functional Representation (FF).

The operations in Table 2 are solved as for BP, since the parentheses are set in BP order. Furthermore, we have shown how to implement operation $level_ancestor$ using the same primitives fwd_search and bwd_search , as well as several others not considered here [27].

The range min-max tree also solves easily operation $rmq(x, y)$, and thus $lca(x, y)$: The area $P[x, y]$ is covered by $O(k \log_k(n/b))$ (super)blocks, which can be scanned in constant time for the minimum excess value. By

storing also the number of minima, we can in a similar way compute $degree(x)$ (which is the number of excess minima in $P[x + 1, findclose(x) - 1]$), $child(x, i)$ (which is the i -th excess minimum in $P[x + 1, findclose(x) - 1]$), and $child_rank(x)$ (which is the number of excess minima in $P[parent(x), x - 1]$).

Implementation notes. The number of minima in each (super)block is stored as described for $m[i]$ and $M[i]$, thus using other $\approx 3.52\%$ overhead. Other small-space structures can be added to support further operations [27].

Labeled trees. We use $L[1, n]$ as for BP. Yet, both binary (using $child(x, i)$) and sequential search are possible with the RMM. We test both.

4 Experimental Comparison.

We performed our experiments on trees coming from (a) LZ78 parsings of text collections, (b) suffix trees of text collections, and (c) the tag structure of XML collections. For cases (a) and (b) we used 200 MB and 50 MB (respectively) prefixes of the the DNA, protein and XML texts from the Pizza&Chili corpus (<http://pizzachili.dcc.uchile.cl>). The resulting trees have 16,373,737, 30,080,186, and 11,775,736 nodes respectively for case (a), whereas for (b) they have 75,095,011, 72,396,959 and 70,723,755 nodes. For case (c), we used an XMark [28] document of about 558 MB, a Medline document (<ftp://ftp.cogsi.ed.ac.uk/pub/disp/OHSXML.tar.gz>) of about 380 MB, and the XML text form Pizza&Chili, of 200 MB. The resulting DOM tree structures have 14, 238, 042, 3, 966, 917, and 9,560,024 nodes, respectively.

We tested the performance of the tree operations by emulating random root-to-leaf traversals of trees (root itself excluded), where in addition every other child of the current node in the path is visited with a given probability p . We retain the order in which such nodes are visited, so $p = 0.0$ corresponds to a pure random root-to-leaf traversal, $0 < p < 1$ simulates range- or approximate-search traversals, and $p = 1$ simulates a full traversal of the tree. The general behavior is quite similar within each class of trees (a), (b) and (c), so for LZ78 we only show the case of DNA text (and later, once, proteins), while for suffix trees we only show the case of proteins, and for XML the XMark document. Table 3 gives some of their characteristics.

In general, as p grows, a higher proportion of small and nearby subtrees are visited. This translates into more close matching parentheses and more cache hits in BP and DFUDS. As closer parentheses are found faster in both schemes, such structures become faster as p grows. LOUDS, instead, is generally insensitive to

| | LZ78 DNA | LZ78 Protein | S. tree Protein | XML XMark |
|--------------------|--------------|-----------------|--------------------|--------------|
| Size | 16, 373, 737 | 30, 080, 186 | 72, 396, 959 | 14, 238, 042 |
| Height | 64 | 126 | 173 | 14 |
| Avg. leaf depth | 14.21 | 7.79 | 11.15 | 7.76 |
| Max.arity | 16 | 25 | 26 | 127,500 |
| Avg.arity | 2.01 | 2.63 | 2.90 | 2.11 |

Table 3: Basic characteristics of the trees tested.

the subtree sizes, as it proceeds levelwise (indeed, it is slightly slower on deeper levels, as the parent/children are farther away).

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 of Linux kernel.

We first measure the times of the core operations, which allow predicting the performance of many operations in Table 2. Some specific tree operations are studied later.

4.1 Core Operations.

Figure 1 shows the space/time tradeoffs achieved for the core parenthesis and bitmap operations, for the cases $p = 0.0$ and $p = 0.2$. The times are averaged over at least 20,000 nodes (for $p = 0.0$), and up to 20 million for larger p values. We show the times for $findclose$ and $enclose$. For the RP representation we use block sizes $b = 32, 64, 128, 256, 512$. For HB we use block sizes $b = 128, 256, 512, 1024, 2048$. For the RMM representation we set $b = 512$ and $k = 8, 16, 32, 64, 128$ (other values of b did not yield better results). These combinations yield the observed space/time tradeoffs.

Modifier “DFUDS” means that the parentheses sequence used is that of DFUDS rather than BP. RP-DFUDS was very close to RP, which is clearly the worst performer as we comment soon, so we omitted it to lighten the plots. On XMark, BP-DFUDS needs more than 9 bits per node and thus it does not appear in the XML plots. We also omitted RMM-DFUDS from all these plots as it was indistinguishable from RMM.

We also show the time for $rank$ and $select$, which offers its own space/time tradeoff (with the same space one solves both) and $prev, next$, which is drawn as a line because it uses brute force and hence does not need any extra space. Operations $prev, next$, were tested over the nodes traversed in the DFUDS sequence (which is equivalent to having chosen LOUDS), as these are the representations where such operations are applied.

The times for $findopen$ are similar to those for $findclose$ on BP. On DFUDS, instead, they are asymmetric: $findclose(x)$ is usually faster as it returns a typ-

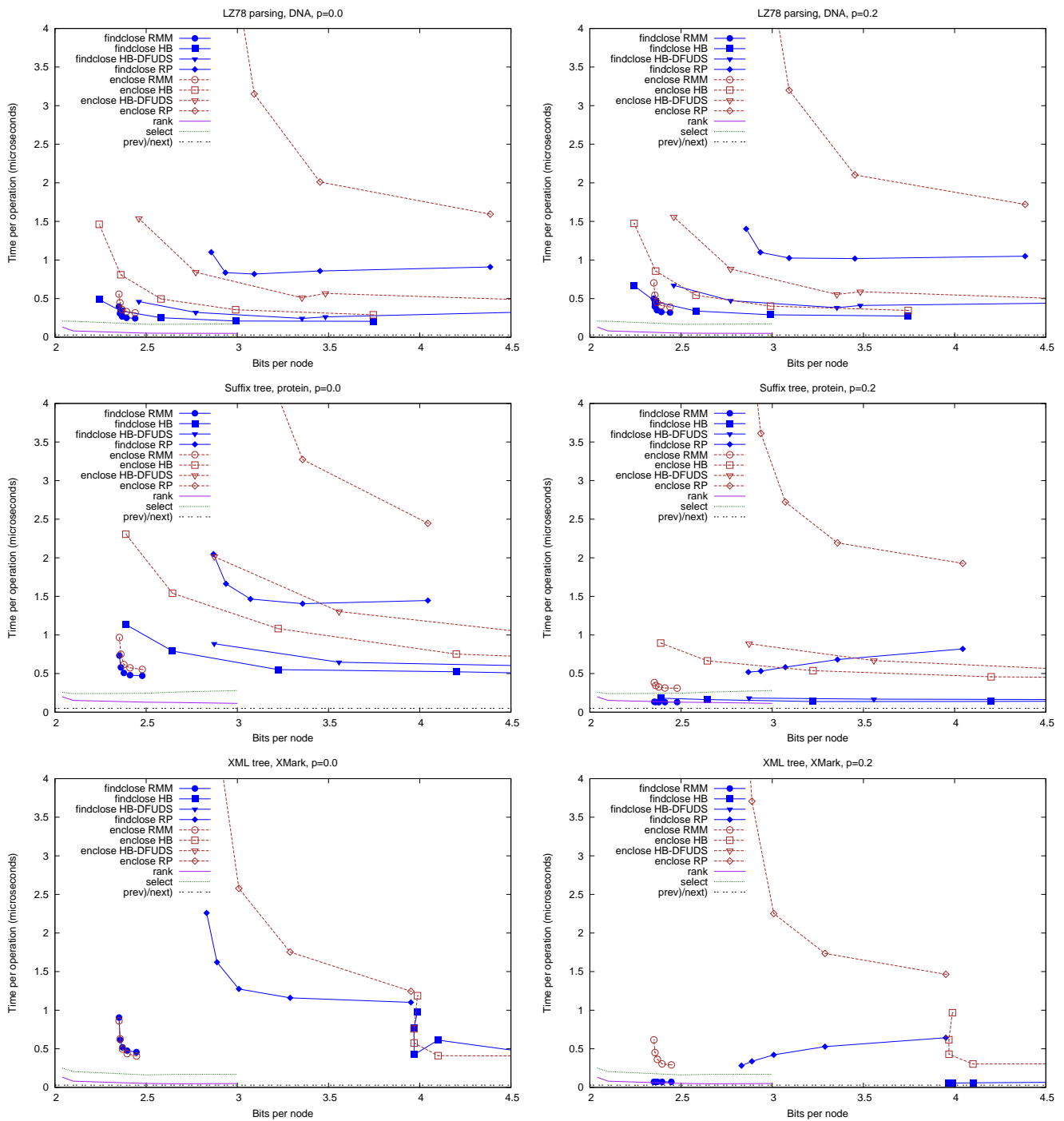


Figure 1: Space/time for the core operations on parenthesis structures, for $p = 0.0$ (left) and $p = 0.2$ (right).

ically closer node. Yet, the operations $findopen(x - 1)$ of Table 2 have a cost very similar to $enclose(x)$ (which usually returns $findopen(x - 1) - 1$).

Clearly RMM (and RMM-DFUDS) is in general the best performer, both in space and time. HB is generally able of approaching its time, yet the space required by HB varies widely depending on the tree shape. On DNA, where the arity is low, HB can improve upon RMM in space, while being competitive in time (at least for $findclose$). In proteins, where the arity is larger, many parentheses become far (especially for $enclose$) and the space of HB cannot anymore reach that of RMM. The effect is dramatic on XMark, where the arity is much higher and HB is unable of operating with less than 100% space overhead over the $2n$ bits. In the DFUDS variants of HB the parentheses tend to be farther than on BP, thus HB-DFUDS requires more space than HB(-BP). In XMark, HB-DFUDS has a space overhead of more than 450%.

This shows that the lack of space guarantees of HB is indeed a problem in practice, and emphasizes the importance of offering such guarantees in fundamental data structures, where one can hardly predict the type of instances that will be faced.

The time performance of RP is clearly the worst. RP is affected by the operations on the compressed bitmap, which in practice pose a significant time overhead, as well as space redundancy over the entropy. This is illustrated, in particular, by the fact that, on $findclose$ for $p > 0$, where many small subtrees are handled, RP in many cases worsens in time when using more space. The reason is that, when the answer is not too far away, it is better to find it sequentially through sparser blocks than to use the heavy machinery of the compressed bitmaps.

Finally, note that the extra space for $rank/select$ is the only one required by LOUDS, and that it works well within just 2% of extra space. This means that, within the small set of operations it supports, LOUDS is competitive in time and better in space than RMM. Note in particular that $select$ may worsen when using more space. This is because the binary search is done over a denser (and larger) set of samples, whose impact on the locality is not recovered by scanning a shorter segment of the bitmap (as this scan is cache-friendly).

Figure 2 shows the extreme case of $p = 1.0$, which simulates a full tree traversal. Here most of the extra structures are useless, as the majority of the operations are carried out on very small trees, where they are easily solved by brute force. In this case LOUDS representation (that is, $rank/select$) is not that fast compared to the others, as this kind of locality does not show up in its levelwise traversal.

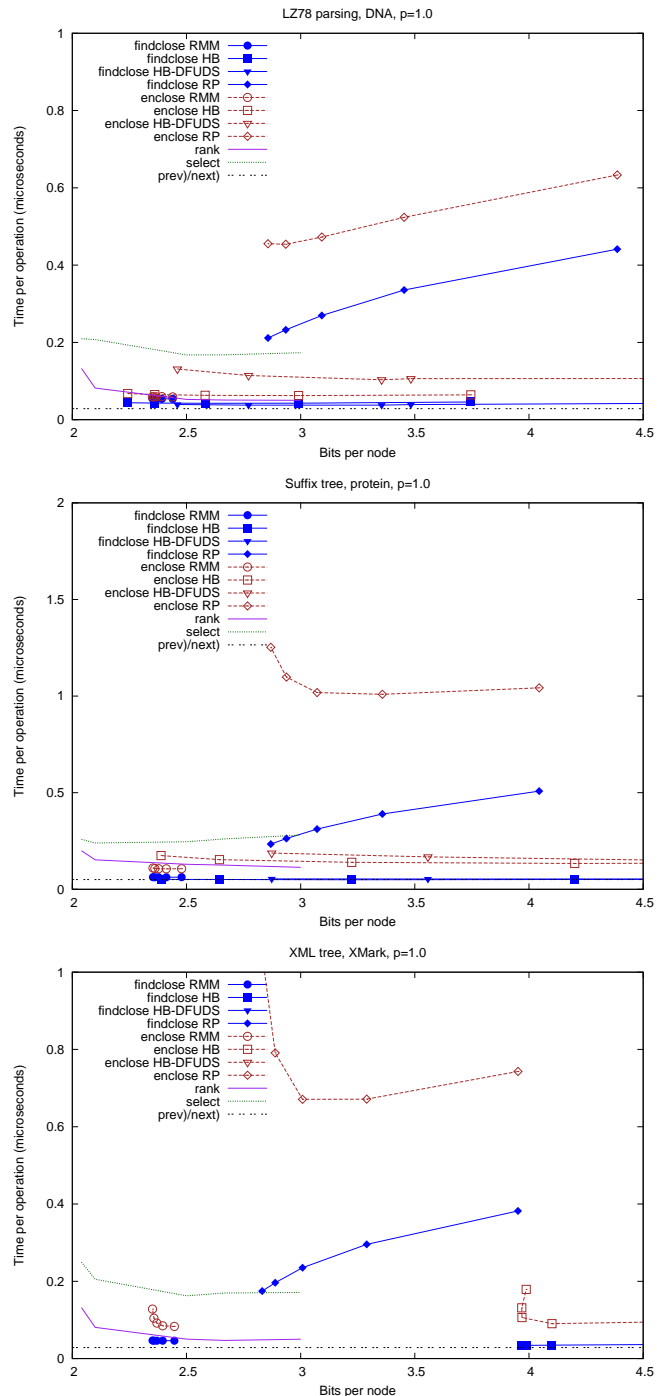


Figure 2: Space/time for the core operations on parenthesis structures, for $p = 1.0$.

| Repres. | Params. | LZ78 DNA | LZ78 Protein | S. tree Protein | XML XMark |
|-----------------------|------------------------|-------------|-----------------|--------------------|--------------|
| FF-BP and FF-DFUDS | $b = 512,$ $k = 32$ | 2.37 | 2.37 | 2.38 | 2.37 |
| HB-BP | $b = 512$ | 2.58 | 2.75 | 3.22 | 3.97 |
| HB-DFUDS | $b = 512$ | 3.35 | 4.17 | 4.84 | 9.52 |
| RP-BP | $b = 64$ | 3.45 | 3.33 | 3.36 | 3.29 |
| LOUDS | $s = 640$ | 2.10 | 2.10 | 2.10 | 2.10 |

Table 4: Default space usage for the representations in Section 4.2.

4.2 Tree Operations.

We tested several operations, a few to show that their times can be predicted with those of the core operations, and most because they are harder to predict: $parent(x)$, $child(x, i)$, $labeled_child(x, s)$, $level_ancestor(x, d)$, and $lca(x, y)$. We choose one representative tree for each operation, as the conclusions are roughly the same for the others. Except for $lca(x, y)$, we chose one reasonable space/time point per structure. These choices are given in Table 4 (recall that FF-* corresponds to RMM).

Figure 3 gives the times for operation $parent(x)$ on the suffix tree for proteins. We use the same sampling method (i.e., via tree traversals) of the core operations, with varying p . As in these, we average the times over at least 20,000 nodes. FF-BP outperforms the other alternatives except for $p = 0.0$, where LOUDS is faster. For higher p (where more smaller subtrees are chosen) LOUDS does not benefit from locality, as explained. The difference between HB-DFUDS and HB-BP is small, and can be attributed to the extra $prev$ operation. The difference between FF-DFUDS and FF-BP is larger.

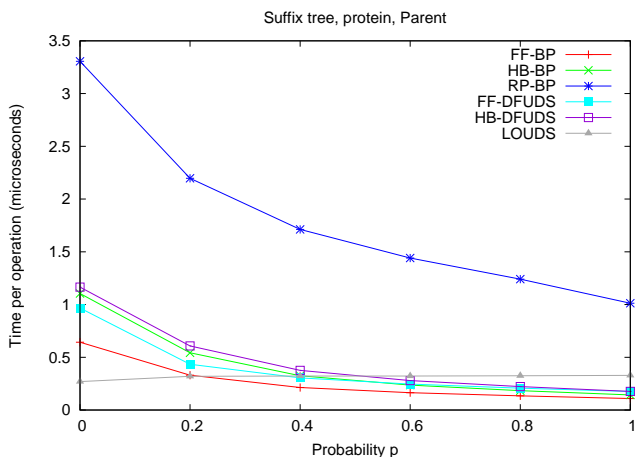


Figure 3: Performance for operation $parent(x)$ as a function of p .

Figure 4 shows the times for operation $child(x, i)$

on the suffix tree of proteins, for $1 \leq i \leq 5$. We choose nodes from the same traversal done for $p = 0.0$ on the core operations, keeping all nodes with degree at least 5. This gives us a sampling of 14,000 nodes to average over. This time we have also included the times over a pointer-based uncompressed tree, which will be discussed in Section 4.4. That time is essentially that of a single memory access, and is close to the time of $first_child(x) = child(x, 1)$ on BP representations. For larger i , as HB-BP and RP-BP operate by brute force, they are essentially $O(i)$, though with different constants (related to their time to carry out $findclose$). LOUDS is clearly $O(1)$ for this operation, and faster than all the others except for the trivial case $i = 1$ (which, on the other hand, is the dominant case on various common traversals). HB-DFUDS is also $O(1)$ in theory (as it always needs one $findclose$ operation), yet it is faster for smaller i values because the answer is closer in those cases. The time of FF-DFUDS is in practice logarithmic on the distance of the answer, as it climbs the range-min-max tree. FF-BP uses its own algorithm for $child$, and its time is also logarithmic on the distance of the answer (and thus very similar to FF-DFUDS). This explains the slow growth of these alternatives. All the three also get closer to LOUDS for larger p (this plot is for $p = 0.0$, as explained), finally surpassing it around $p = 0.6$.

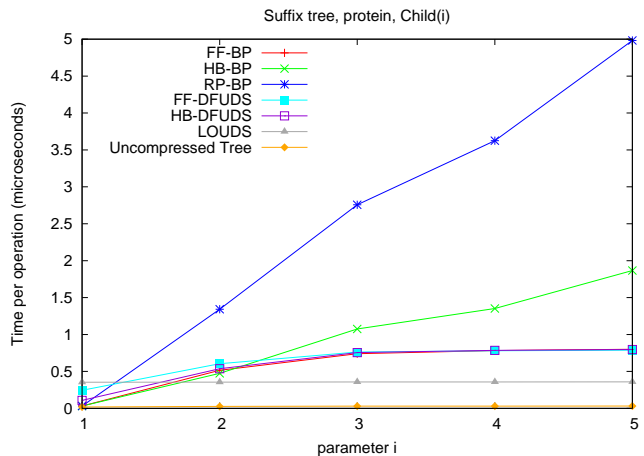


Figure 4: Performance for operation $child(x, i)$ as a function of i .

Figure 5 illustrates the performance for operation $labeled_child(x, s)$ on the LZ78 parsing of proteins. We use exactly the same method of sampling nodes of $child(x, i)$, and also plot the times as a function of i , so this time the argument s is the one resulting in descending to the i -th child. This is an operation where LOUDS excels, DFUDS does well, and FF-BP falls far

behind. The binary search of LOUDS and DFUDS is done directly over a range in L , which makes them very fast, much faster than the linear searches of BP alternatives (still DFUDS and LOUDS need to finish with one $child(x, i)$ operation after finding i). The time differences among LOUDS and DFUDS alternatives are lower than for $child(x, i)$ because all must carry out the same binary search. On the other hand, the linear times of the BP alternatives are slightly costlier than for $child(x, i)$ because, although they probe the same nodes, they must carry out one further $rank$ operation per node to find the letter in L . They are only faster in the lucky case of s labeling the first child. Finally, FF-BP is surprisingly slow. Although it carries out a binary search, this search is not local and requires various $child(x, i)$ operations, which renders it slower than some linear searches in practice. Indeed, it is much better to linearly scan for the i -th child using $findclose$, as illustrated by FF-Seq-BP, which is the fastest among the $O(i)$ methods.

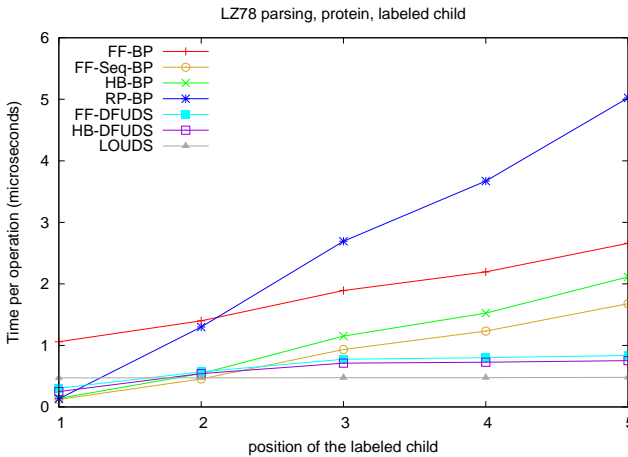


Figure 5: Performance for operation $labeled_child(x, s)$ as a function of i , the position of the resulting child.

Figure 6 compares the performance for operation $level_ancestor(x, d)$ on the LZ78 tree of DNA, for $1 \leq d \leq 5$. We choose all the nodes from the sampling of core operations corresponding to $p = 0.0$, and keep the nodes with depth at least 5. This leaves us 2,000 nodes to average over. Here all the techniques except FF-BP use brute force (note FF-DFUDS cannot benefit from the FF-BP algorithm, as excesses do not correspond to depths in DFUDS). The times, consequently, are basically linear, except for FF-BP, whose algorithm based on operation bwd_search is logarithmic in the distance to the answer. This dominates all the other costs and shows its practicality. Note the times are about 50% faster than the corresponding to $parent(x)$

on the suffix tree of proteins, as these nodes have fewer children and hence their parents are closer.

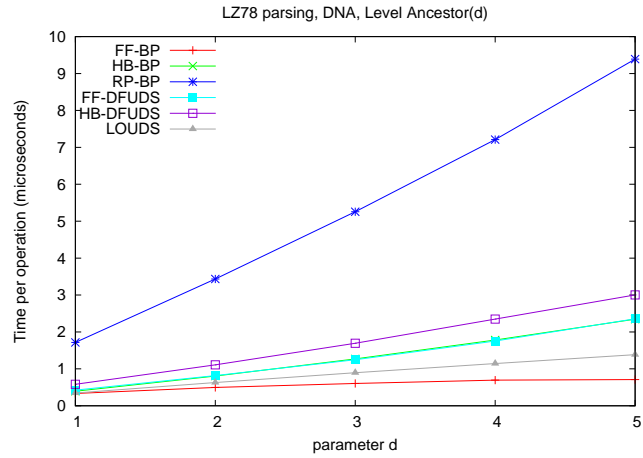


Figure 6: Performance for operation $level_ancestor(x, d)$ as a function of d .

Finally, Figure 7 shows operation $lca(x, y)$ on XMark, for different memory usage of the structures. This time we simply chose 100,000 pairs of tree nodes at random. This operation is implemented by brute force on all representations except FF-BP and FF-DFUDS, which provide a native solution to lca . It can be seen that FF and LOUDS (with its lightweight brute-force algorithm) sharply dominate the space/time tradeoff. FF-DFUDS is slightly slower than FF-BP because it needs an extra check that invokes an $ancestor$ query. Using the RMQ-based solution for the other alternatives would achieve competitive times (0.9 microseconds according to a simple experiment we carried out), yet it would require including the extra constant-time RMQ structure [9], which adds $6.12n$ bits to the already large HB and RP representations.

4.3 Asymptotics.

Our experiments have considered fixed trees. While all the operations are in theory constant-time, in practice the implementation of $select$ is $O(\log n)$, as well as our FF operations¹. Furthermore, cache effects can significantly affect the performance. Thus, one may wonder how the tree size n affects the results given.

Figure 8 shows the times for operations $findclose$ and $enclose$ as a function of the tree size n . For this sake, we created random binary trees of increasing size, from 50 to 350 million nodes, and tried the BP alternatives

¹While the theoretical article [27] achieves constant time, our FF implementation builds solely on the RMM, which seems much more practical but requires time logarithmic on the tree size.

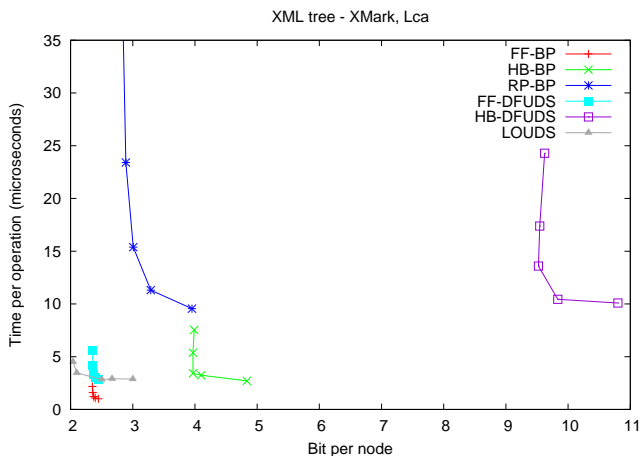


Figure 7: Performance for operation $lca(x, y)$ as a function of the space usage.

on them. We maintained the space usage parameters of Table 4, obtaining 2.37–2.38 bits per node for RMM, 2.62–2.99 for HB, and 3.68–3.74 for RP. The nodes to operate are collected over 100,000 random root-to-leaf traversals (akin to $p = 0.0$ above).

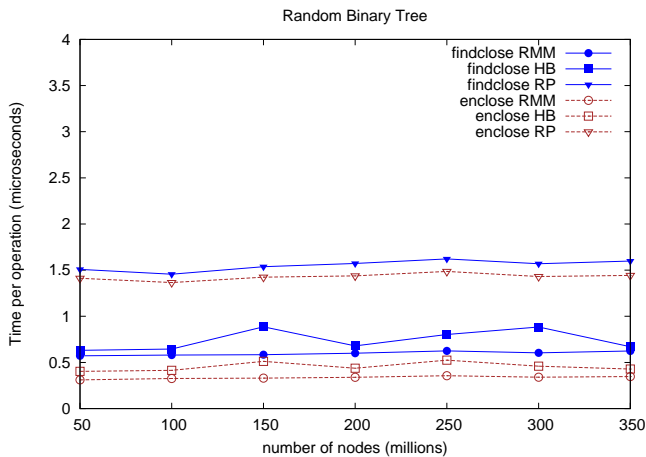


Figure 8: Performance as a function of the tree size, for random binary trees.

As it can be seen, apart from some oscillations, there is no any clear increase in the time as we handle larger trees. This is due in part to the fact that many operations (most clearly those of RMM) depend not on the total tree size, but on the distance between the query and the answer node, and this average distance varies very mildly on n even for the traversal with $p = 0.0$.

This result confirms the scalability of the solutions and the stability of our results on larger trees. Note in passing that $findclose(x)$ is slower than $enclose(x)$ on

binary trees, as half the times the answer of the latter is simply $x - 1$.

4.4 Non-Succinct Tree Representations.

A natural question is how the times we have shown (usually below the microsecond per operation, for the best performing structures) compare to a plain pointer-based tree representation. For this sake, we set up such a tree, where each node has a pointer to an array of its children. This allows one to implement $child(x, i)$ using just one memory access. This takes around 18–31 nanoseconds in our machine and is illustrated in Figure 4.

This means that an uncompressed tree, in the cases where it can actually fit in main memory, is about 12–20 times faster than LOUDS, our best compressed representation for this operation. If we consider all the operations, and take FF-BP as the alternative (as it gives the widest support), we have that in most cases the time of FF-BP is within half a microsecond (30 times slower than the single memory access needed by the uncompressed representation) and for some complex operations, up to 2 microseconds (120 times slower).

Still, it is important to illustrate in numeric terms what it means to be succinct or plain, especially when we need more functionality than moving to a child. A plain representation on a 32-bit machine spends 32 bits per node (that is, 13 times more than FF-BP and 15 more than LOUDS), if we arrange the tree in levelwise order on an array, and the cell of each node points to the cell of its first child. This arrangement supports efficiently the most basic navigation: $child$, $degree$, $isleaf$, $first_child$, and $next_sibling$. However, each other operation we wish to support efficiently requires another integer: $parent$, $depth$, $subtree_size$, pre_rank , etc. Thus, just supporting these four operations we would require 160 bits per node. The more functionality we wish to support, the less likely is that the pointer-based tree will fit in main memory, in applications managing massive trees. In contrast, the powerful FF-BP representation supports all those operations within the same space (less than 2.4 bits per node).

The more sophisticated operations, like lca and $level_ancestor$, deserve a special mention. They can be solved within $O(n)$ -word space and $O(1)$ time, but they require several (not just 1) further integers per node. Worse than that, they require several memory accesses. For example, the classical lca algorithm [2] requires 7 extra integers (224 bits) per node and 12 non-local memory accesses. Thus its expected time is around 0.2–0.4 microseconds, whereas FF-BP solves lca within the microsecond (just up to 5 times slower) and the same 2.4 bits per node.

| Repres. | LZ78 DNA | LZ78 Protein | S. tree Protein | XML XMark |
|----------------|-------------|-----------------|--------------------|--------------|
| gaps | 1.62 | 1.46 | 1.36 | 1.59 |
| γ -code | 2.24 | 1.92 | 1.72 | 2.18 |
| real γ | 3.43 | 2.72 | 2.44 | 2.90 |

Table 5: Space usage for the larger representations with minimum functionality. Only the last column is a real encoding scheme.

On the other hand, there are intermediate tree representations [16, 5, 14] which, although do not guarantee $2n + o(n)$ bits of space usage (only $O(n)$ bits in general), offer a relevant tradeoff in practice. Translated into a general tree representation, they usually boil down to a scheme as follows: Arrange the nodes of the tree on an array $T[1, n]$ by performing a preorder traversal. The cell of each node stores where is its next sibling. Unlike our previous uncompressed scheme, akin to a LOUDS layout, this resembles more BP: we immediately have $first_child(i) = i + 1$ and $next_sibling(i) = T[i]$ (yet, as before, we lack most of the other operations unless we pay for the space of storing the answers explicitly). By storing the pointers in relative form and using a variable-length encoding, compression is possible since most of the subtrees are small. The value of the relative pointer from node x to its next sibling will be just $subtree_size(x)$.

To measure the practical performance of this idea, we computed $\sum_{x \in T} \lceil \log(subtree_size(x) + 1) \rceil$ on our trees, which gives a lower bound to the size that can be achieved by this technique. It is shown in the first line of Table 5. It is a lower bound because (i) storing the exact number of bits the number needs is not possible (a representation like, say, γ -encoding, is necessary); and (ii) since a variable-length code is used, the pointers must point to bit offsets, not to cell numbers of T .

The second row of Table 5 addresses problem (i) using γ -codes, which gave us good results. It shows the value $\sum_{x \in T} |\gamma(subtree_size(x))| = \sum_{x \in T} (2 \lceil \log(subtree_size(x) + 1) \rceil - 1)$. The last line of the table addresses problem (ii), only now reaching a real solution to the problem. It γ -encodes not the subtree size of x , but the length of the representation of the subtree of x (which recursively uses γ -codes, a leaf using $|\gamma(1)| = 1$ bit). One adds this value to the pointer to x and reaches the bit-offset of its next sibling.

As it can be seen, these representations are space-efficient in practice, competing with our $2.4n$ -bit representations, and possibly being very fast for operations $first_child$, $next_sibling$, and $isleaf$, that is, for a full tree traversal. Yet, as we have seen in Figure 2, one can do pretty well with the plain $2n$ parentheses for such

traversal. Still, assuming that $next_sibling$ will be significantly faster by decoding a γ -code, the representation offers a rather limited repertoire of operations. By somehow mixing fixed- and varying-length representations one could include $child$ and $degree$, at some price in space. Yet, as before, each further operation to support requires storing extra data that is already included in our $2.4n$ -bit representations (and many are also included in our $2.1n$ -bit LOUDS format).

This shows that the key advantage in the modern succinct tree representations is not their $2.x$ -bit space usage, as this is not that difficult to achieve. The key advantage is the wide functionality these succinct representations support within this space.

5 Conclusions.

We have carried out a rather exhaustive comparison of the best techniques to represent general trees of n nodes in little space. We focused on the most succinct schemes, which use $2n + o(n)$ bits. It turns out that the recent so-called fully-functional (FF) representation [27] offers an excellent combination of space usage, time performance, and functionality. It implements a large set of simple and sophisticated operations, and in most cases it outperforms all the other structures in time and space. For example, with less than 20% overhead over the minimum $2n$ bits of space, it carries out all the operations within the microsecond (up to 2 microseconds for a few of them). Indeed, we have recently used FF representation successfully for an application managing large XML databases in main memory while supporting fast XPath operations [1].

In some applications requiring limited functionality, other structures are of interest. LOUDS can implement a more reduced set of navigation operations using just 5% extra space (and even 2%, at a small price in time) over the $2n$ bits. Within that little space, LOUDS is very competitive in time, being the fastest choice in several cases. This is particularly noticeably on large-arity trees and on random root-to-leaf traversals, while the other schemes tend to perform better on lower arities or when many nearby nodes have to be processed (in particular, for full traversals).

A case where LOUDS excels and FF falls far behind is for descending to a child given its label. Here DFUDS is a good alternative, only somewhat slower than LOUDS. Although DFUDS variants are usually a bit slower than their BP counterparts, FF-DFUDS can be a reasonable compromise when this labeled-child operation is important and full functionality is desired. It requires the same space of FF(-BP).

We believe in particular that LOUDS representation has not been sufficiently studied in the litera-

ture because it lacks support for some sophisticated operations, even if it offers a very attractive solution for the simplest (and most common) navigation operations. These are already richer than the minimal set of downward-traversal operations that “classical” compressed representations based on variable-length encoding offer. In this paper we have introduced novel practical solutions for several sophisticated operations, which turn out to be surprisingly competitive (others, such as computing the subtree size, are still relatively too slow). We believe that even more could be done.

An important topic of future work is to achieve practical succinct dynamic tree representations, where nodes can be added to and deleted from the tree. In this aspect even the theoretical proposals are lacking. The FF representation admits a simple and practical dynamic implementation with $O(\log n)$ time for all the operations, and a more complex theoretical one reaching the lower bound $O(\log n / \log \log n)$ [26].

Our implementations have been left public in *Google Code*, extending the current `libcds` library, to foster their use in diverse applications and their comparison to other practical proposals to come.

References

- [1] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, and N. Välimäki. Fast in-memory XPath search over compressed text and tree indexes. In *Proc. 26th ICDE*, 2010. To appear.
- [2] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th LATIN*, LNCS 1776, pages 88–94, 2000.
- [3] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- [4] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
- [5] D. Blandford, G. Blelloch, and I. Kash. An experimental analysis of a compact graph representation. In *Proc. 6th ALENEX*, pages 49–61, 2004.
- [6] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.
- [7] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. 5th WEA*, pages 134–145. LNCS 4007, 2006.
- [8] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th SWAT*, LNCS 5124, pages 173–184, 2008.
- [9] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, LNCS 4614, pages 459–470, 2007.
- [10] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.
- [11] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th SODA*, pages 1–10, 2004.
- [12] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. Rao. On the size of succinct indices. In *Proc. 15th ESA*, pages 371–382. LNCS 4698, 2007.
- [13] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th WEA (posters)*, pages 27–38, 2005.
- [14] A. Gupta, W.-K. Hon, R. Shah, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. 5th WEA*, pages 158–169, 2006.
- [15] M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th ICALP*, LNCS 4596, pages 509–520, 2007.
- [16] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
- [17] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th SODA*, pages 575–584, 2007.
- [18] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms (TALG)*, 4(3):article 28, 2008.
- [19] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- [20] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
- [21] I. Munro and S. Rao. Succinct representations of functions. In *Proc. 31th ICALP*, LNCS 3142, pages 1006–1015, 2004.
- [22] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithms (JEA)*, 13(article 2), 2009. 49 pages.
- [23] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, 2007.
- [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] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [26] K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768, 2009. <http://arxiv.org/abs/0905.0768>.
- [27] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st SODA*, 2010. To appear.
- [28] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. 28th VLDB*, pages 974–985, 2002.