Available online at www.sciencedirect.com

ScienceDirect

Theoretical Computer Science 368 (2006) 231–246

Theoretical Computer Science

www.elsevier.com/locate/tcs

A simple optimal representation for balanced parentheses

Richard F. Geary[a], Naila Rahman[a], Rajeev Raman[a,*], Venkatesh Raman[b]

[a] *Department of Computer Science, University of Leicester, Leicester LE1 7RH, UK*
[b] *Institute of Mathematical Sciences, Chennai 600 113, India*

Abstract

We consider *succinct*, or highly space-efficient, representations of a (static) string consisting of n pairs of balanced parentheses, which support natural operations such as finding the matching parenthesis for a given parenthesis, or finding the pair of parentheses that most tightly enclose a given pair. This problem was considered by Jacobson [Space-efficient static trees and graphs, in: Proc. of the 30th FOCS, 1989, pp. 549–554] and Munro and Raman [Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (2001) 762–776] who gave $O(n)$-bit and $2n + o(n)$-bit representations, respectively, that supported the above operations in $O(1)$ time on the RAM model of computation. This data structure is a fundamental tool in succinct representations, and has applications in representing suffix trees, ordinal trees, planar graphs and permutations.

We consider the practical performance of parenthesis representations. First, we give a new $2n + o(n)$-bit representation that supports all the above operations in $O(1)$ time. This representation is conceptually simpler, its space bound has a smaller $o(n)$ term and it also has a simple and uniform $o(n)$ time and space construction algorithm.

We implement our data structure and a variant of Jacobson's, and evaluate their practical performance (speed and memory usage), when used in a succinct representation of trees derived from XML documents. As a baseline, we compare our representations against a widely used implementation of the standard DOM (document object model) representation of XML documents. Both succinct representations use orders of magnitude less space than DOM and tree traversal operations are usually only slightly slower than in DOM.

Keywords: Succinct data structures; Parentheses representation of trees; Compressed dictionaries; XML DOM

1. Introduction

Given a static balanced string of $2n$ parentheses, we want to represent it *succinctly* or space-efficiently, so that the following operations are supported in $O(1)$ time on the RAM model:

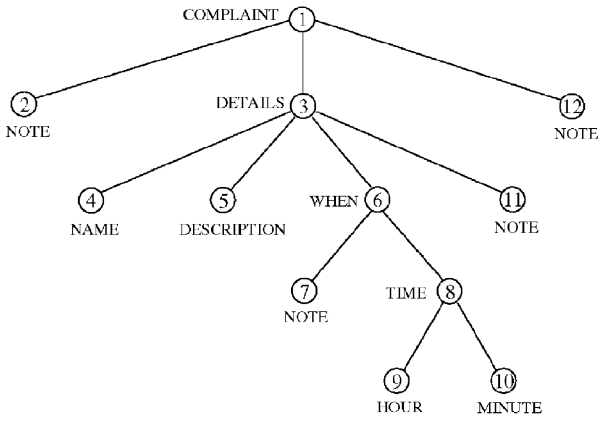
- FINDOPEN(x), FINDCLOSE(x): To find the index of the opening (closing) parenthesis that matches a given closing (opening) parenthesis x.
- ENCLOSE(x): To find the opening parenthesis of the pair that most tightly encloses x.

By counting the number of balanced parenthesis strings, one can see that the string requires $2n - O(\lg n)$ bits in the worst case, so a naive representation of the string is very close to optimal in terms of space usage. However, the above

* Corresponding author.
E-mail addresses: geary@mcs.le.ac.uk (R.F. Geary), naila@mcs.le.ac.uk (N. Rahman), rr29@mcs.le.ac.uk (R. Raman), vraman@imsc.res.in (V. Raman).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.09.014

```
<COMPLAINT>
   <NOTE></NOTE>
   <DETAILS>
      <NAME></NAME>
      <DESCRIPTION></DESCRIPTION>
      <WHEN>
         <NOTE></NOTE>
         <TIME>
            <HOUR></HOUR>
            <MINUTE></MINUTE>
         </TIME>
      </WHEN>
      <NOTE></NOTE>
   </DETAILS>
   <NOTE></NOTE>
</COMPLAINT>
```



parenthesis representation:

(( ) (( ) ( ) (( ) (( ) ( ))) ( )) ( ))

1 2 3 4 5 6 7 8 9 10 11 12

nodeName array:

[0, 1, 2, 3, 4, 5, 1, 6, 7, 8, 1, 1]

nodeName dictionary:

0 = COMPLAINT	5 = WHEN
1 = NOTE	6 = TIME
2 = DETAILS	7 = HOUR
3 = NAME	8 = MINUTE
4 = DESCRIPTION	

Fig. 1. Top left: Small XML fragment (only tags shown). Top right: Corresponding tree representation. Bottom: Succinct representation of document.

operations would essentially take linear time to support. One way to support O(1)-time operations is to note that the string is static and precompute and store answers for all possible arguments, but this uses $O(n \lg n)$ bits, $\Theta(\lg n)$ times more space than necessary. Jacobson [14] and Munro and Raman [20] gave $O(n)$-bit and $2n + o(n)$-bit representations, respectively, that supported the above operations in O(1) time on the RAM model of computation. [1] Parenthesis representations are fundamental to succinct data structures, and have applications to suffix trees [22,18], *ordinal* trees [2,3,20,11], k-page graphs [14,20] and stack-sortable permutations [17]. A topical motivation, and the starting point of our work, is the use of this data structure in the representation of (large, static) XML documents. The correspondence between XML documents and ordinal trees is well-known (see e.g. Fig. 1). In this paper we consider simplified XML documents, where we ignore a number of secondary features, [2] and also assume that the document consists purely of markup (i.e. there is no free text).

The XML document object model (DOM) [15] is a standard interface through which applications can access XML documents. DOM implementations store an entire XML document in memory, with its tree structure preserved. At the heart of DOM is the `Node` interface, which represents a single node in the tree. The node interface contains attributes such as `nodeName`, `nodeValue` and `nodeType` to store information about the node, as well as `parentNode`, `firstChild`, `lastChild`, `previousSibling` and `nextSibling`, which act as a means to access other related nodes. The usual, but naive, way of implementing the DOM is to store with each node a pointer to the parent, the first/last child, and the previous/next sibling. Unfortunately, this can take up many times more memory than the raw XML file. This 'XML bloat' significantly impedes the scalability and performance of current XML query processors [1], especially if the DOM representation does not fit in main memory (which can happen for fairly modest-sized documents).

To represent XML documents succinctly, while providing the essential features of the `Node` interface, we store the tree as a sequence of parentheses, identifying nodes with the position of their open parentheses. We also store a sequence of values $\sigma_1, \ldots, \sigma_n$, where σ_i is the tag of the ith node in pre-order (see Fig. 1); other information associated with the node can be stored analogously. Given an open parenthesis that represents a node v, in order to access the tag

[1] Jacobson's result was stated for the bit-probe model, but it can be modified to run in O(1) time on the RAM model [19].

[2] Such as: attributes and their values, namespace nodes, comments, etc.

name and other information associated with v, we augment the parenthesis data structure with a standard data structure that, for any position i in the sequence, gives the number of open parentheses in positions $1, \ldots, i$, and occupies $o(n)$ bits [19]; this gives the pre-order number of the node whose open parenthesis is at position i in the parenthesis string.

The viability of such a representation depends crucially on the speed and space-efficiency of the parenthesis data structure. A good implementation must find the right trade-off between storing pre-computed data—the "insignificant" $o(n)$ terms can easily dominate space usage—and computation time. There has been work on implementations of space-efficient trees, including k-ary trees, where each edge from a node to its children is labelled with a distinct letter from an alphabet [8] and Patricia trees [7] among others. Chupa, in unpublished work, described an implementation of a restricted static binary tree representation [6]. Compressed self-indexing dictionaries have been implemented in [9,12]. We are not aware of any implementations of a parenthesis data structure.

We begin by giving a new, conceptually simple, $2n + o(n)$-bit parenthesis data structure. Our new data structure uses no complex subroutines (e.g. [20] use perfect hash tables) and it has a lower order term in the space usage of $O(n \lg\lg n / \lg n)$ bits versus $\Theta(n \lg\lg\lg n / \lg\lg n)$ bits in [20]. It also has a simple and uniform $o(n)$-time and space construction algorithm, which is not known of the data structure of [20]. Indeed, to achieve $O(n)$ construction time, [20] needs to use either randomisation, or a recent complex algorithm [13] for constructing perfect hash tables.

We implement a version of Jacobson's data structure as well as the new one, evaluating their space usage and speed. As a baseline, we also compare with CenterPoint XML [4] which is an open-source C + + XML DOM library. The standard test we perform with an XML document is to perform a traversal (depth-first and breadth-first) of the tree, both in a standard DOM implementation and in our representation, counting the number of nodes of a given type (this is a fairly canonical operation in manipulating XML documents). As expected, both succinct schemes use orders of magnitude less space than Centerpoint XML—it is surprising how modest the computational overhead of the succinct schemes is.

2. A simple parenthesis representation

Both Jacobson's and Munro and Raman's representations divide the given string of parentheses into equal-sized blocks of B parentheses, and identify a set of $O(n/B)$ parentheses as *pioneers*. They explicitly keep the position of the matching parentheses of the pioneer parentheses. They also store enough other information with blocks and/or with individual parentheses to detect pioneer parentheses, as well as to find the match of any parenthesis, from the position of the match of its closest pioneer parenthesis. They also store a small number of tables, typically, to find answers within a block.

Jacobson takes $B = \Theta(\lg n)$ and so the number of pioneer parentheses is $O(n/\lg n)$. He stores essentially the location of the matching parenthesis for each pioneer explicitly. He uses a bit vector (along with $O(n)$ bits of auxiliary storage) to detect pioneer parentheses, and keeps the *excess*—the number of open minus the number of closing parenthesis—at each block boundary. Each of the above takes $\Theta(n)$ bits, and so the overall space bound is also $\Theta(n)$ bits. In order to reduce the space bound to $2n + o(n)$ bits, Munro and Raman employ a three level blocking scheme (big, small and tiny), using blocks of $\Theta(\lg^2 n)$, $\Theta((\lg\lg n)^2)$ and $\Theta(\lg\lg n)$, respectively, storing auxiliary data at each level. In particular, they store the positions of $\Theta(n/(\lg n)^2)$ pioneer parentheses (with respect to big blocks) in a perfect hash table. Constructing this perfect hash table takes $O(n)$ expected time and space [10] or $O(n)$ time using the rather complex algorithm of [13].[3] The need to store (slightly different) auxiliary information at different block sizes contributes both to the implementation complexity and to the lower-order term in the space bound (the latter is important in determining space usage in practice).

Our representation also divides the given parenthesis string into blocks of size $\Theta(\lg n)$. We modify the definition of a pioneer so that the sequence of pioneer parentheses is itself a balanced string of $O(n/\lg n)$ parentheses. Our representation is based on three main observations. First, the positions of the sequence of pioneer parentheses can be stored using $o(n)$ bits using a *fully indexable dictionary* (FID) [21]. Second, representing the string of pioneer parentheses recursively gives enough information to support the basic operations in constant time. (Recursing at most twice, we have a set of $O(n/\lg^2 n)$ pioneer parentheses, which is small enough that it can be stored using the trivial

[3] One needs to use the result of [13, Section 4] rather than the main result (Theorem 1.1), in order to get a uniform algorithm.

representation.) Third, looking closely at the requirements of the FID, we are able to replace the FID of [21] by a very simple data structure. We now discuss the new parenthesis structure, following the above outline.

2.1. Fully indexable dictionaries

For a positive integer M, let $[M] = \{1, \ldots, M\}$. Given a bit-vector of length M which has 1s at a set of positions $S \subseteq [M]$, $|S| = N$, and zeros elsewhere, we define the operations:
RANK(x, S): Given $x \in [M]$, return $|\{y \in S | y \leqslant x\}|$.
SELECT(i, S): Given $i \in [N]$, return the ith smallest element in S.
We call a representation of S that supports the above two operations in O(1) time a *nearest neighbour dictionary* (*NND*), as the operations below are also supported in O(1) time:
PRED(x, S): Given $x \in [M]$, return x if $x \in S$ and $\max\{y \in S | y < x\}$ otherwise.
SUCC(x, S): Given $x \in [M]$, return x if $x \in S$ and $\min\{y \in S | y > x\}$ otherwise.
An NND that supports RANK and SELECT, on S and $\bar{S}$ simultaneously, where $\bar{S}$ is the complement of S, in O(1) time, has been called a *fully indexable dictionary* (FID) [21]. The following is known about FID (and hence about NND) representations:

Theorem 1 (*Raman et al. [21, Lemma 4.1]*). *There is an FID for a set $S \subseteq [M]$ of size N using at most $\lceil \lg \binom{M}{N} \rceil +$ $O(M \lg\lg M / \lg M)$ bits.*

In particular, we have, from Theorem 1:

Corollary 2. *There is an NND for a set $S \subseteq [M]$ of size $N = O(M/\lg M)$ that uses $O(M \lg\lg M / \lg M) = o(M)$ bits.*

2.2. The new representation

We now assume that we are given a balanced string of $2n$ parentheses and our goal is to support FINDOPEN, FINDCLOSE and ENCLOSE operations in constant time. We now describe the new data structure to store any balanced string of parentheses of length $2N \leqslant 2n$.

If N is $O(n/\lg^2 n)$, then we represent the sequence using the trivial structure which stores the pre-computed answer for each of the operations above, for every parenthesis. This takes $O(N \lg N) = o(n)$ bits. Otherwise, we divide the parenthesis string into equal-sized *blocks* of size $B = \lceil (\lg N)/2 \rceil$. We number these blocks $1, \ldots, \beta \leqslant 4N/\lg N$, and by $b(p)$ we denote the block in which the parenthesis p lies. The matching parenthesis of p is denoted by $\mu(p)$. We call a parenthesis p *far* if $\mu(p)$ is not in the same block as p (and note that $\mu(p)$ is itself a far parenthesis). At any position i, we call the number of open parentheses minus the number of closing parentheses in positions $1, \ldots, i$ as the *left excess* at i. Similarly, we call the number of closing parentheses minus the number of open parentheses in positions $i, \ldots, 2N$ as the *right excess* at i. Consider an opening far parenthesis p, and let q be the far opening parenthesis that most closely precedes p in the string. We say that p is an *opening pioneer* if $b(\mu(p)) \neq b(\mu(q))$ (cf. [14]). The definition of a closing pioneer p is as above, except that q would be the far parenthesis immediately *after* p. A *pioneer* is either an opening or closing pioneer. Note that the match of a pioneer may not be a pioneer itself.

Lemma 3 (*Jacobson [14, Theorem 1]*). *The number of opening pioneers in a balanced string divided into β blocks is at most $2\beta - 3$. The same holds for the number of closing pioneers.*

Proof. The *pioneer graph* which has nodes $1, \ldots, \beta$ and edges $(b(p), b(\mu(p)))$, for all opening pioneers p, is outerplanar and has no parallel edges. Therefore, it has at most $2\beta - 3$ edges. □

For a given block size B, we define the *pioneer family* as the set of all pioneers, together with all their matching parentheses (recall that if p is a pioneer, $\mu(p)$ need not be one). Clearly, the substring comprising only the parentheses in the pioneer family is balanced. We now bound the size of the pioneer family.

Proposition 4. *The size of the pioneer family is at most $4\beta - 6$.*

Proof. The pioneer family graph, defined analogously to the pioneer graph, is itself outerplanar, allowing us to conclude that the pioneer family is of size at most $2 \cdot (2\beta - 3)$ or $4\beta - 6$. □

Remark 5. An alternate characterisation of the pioneer family is as follows. Suppose that we add to the set of pioneers (according to Jacobson's original definition) the leftmost far opening parentheses (if any) in a block, as well as the rightmost far closing parenthesis (if any) in a block. Then the resulting set of parentheses is precisely the pioneer family.

Our structure has the following four parts:

(1) the original parenthesis string π of length $2N$;
(2) an NND (Corollary 2) that stores the set $P \subseteq [2N]$ of the positions in π that belong to the pioneer family;
(3) a recursive parenthesis data structure for the pioneer family; and
(4) a constant number of tables that allow us to operate on blocks in O(1) time. For example, a table that stores for every block b, and for every $i = 1, \ldots, B$, the position of the matching parenthesis of the parenthesis at position i, if the match is inside the block (the table stores 0 if the match is not inside the block). Such tables take at most $O(\sqrt{N}(\lg N)^2) = o(N)$ bits.

We now calculate the space usage. The tables take $O(\sqrt{N}(\lg N)^2)$ bits. Since $|P| \leqslant 16(N/\lg N)$, the NND for pioneers takes $O(N \lg\lg N/\lg N)$ bits by Corollary 2. Thus, if $S(N)$ is the space used by the structure, then $S(N)$ satisfies

$$S(N) = O(N \lg N) \quad \text{if } N \text{ is } O(n/\lg^2 n) \text{ and}$$
$$S(N) = 2N + S(8N/\lg N) + O(N \lg\lg N/\lg N) \quad \text{otherwise.}$$

It is easy to see that $S(n) = 2n + O(n \lg\lg n/\lg n) = 2n + o(n)$ bits.

2.3. Operations

Now we describe how the operations are implemented.

FINDCLOSE(p): Let p be the position of an open parenthesis. First determine by a table lookup whether it is far. If not, the table gives the answer. If it is, use PRED(p, P) to find the previous pioneer p^*. We can show that this will be an open parenthesis. Find its position in the pioneer family using RANK(p^*, P) and find its match in the pioneer family using the recursive structure for P; assume that this match is the jth parenthesis in the pioneer family. We then use SELECT(j, P) to find the position of $\mu(p^*)$ in π. Now observe that since the first far parenthesis in each block is a pioneer, p^* and p are in the same block. Compute i, the change in left excess between p and p^*, using a table lookup. Noting that $\mu(p)$ is the leftmost closing parenthesis in $b(\mu(p^*))$ starting from $\mu(p^*)$, with right excess i relative to $\mu(p^*)$, we locate $\mu(p)$ using a table. FINDOPEN is similar.

ENCLOSE(c): Let p = ENCLOSE(c) such that p and c are both open parentheses. From one (or two) table lookup(s) determine whether either of $\mu(p)$ or p is in the same block as c. If so, we can return p using, if necessary, one call to FINDOPEN. If not, we proceed as follows. Let c' = SUCC(c, P). If c' is a closing parenthesis then let p' = FINDOPEN(c'). Otherwise find the position of c' in the pioneer family using RANK, find the parentheses enclosing c' in the pioneer family and using SELECT translate the result into a parenthesis p' in π. We claim that in both cases $(p', \mu(p'))$ is the pair of pioneer family parentheses that most tightly encloses c. Let q = SUCC($p' + 1$, P). If q is in the same block as p' then p is the first far parenthesis to the left of q. Otherwise, p is the rightmost far parenthesis in the block containing p'. In either case, the answer is obtained from a table.

To prove the correctness of the algorithm, we observe that if p or $\mu(p)$ are in the same block as c, then we can find p using table lookup (and possibly FINDOPEN($\mu(p)$)). Otherwise since both p and $\mu(p)$ are in different blocks to c, $b(p) < b(c) < b(\mu(p))$ and hence both p and $\mu(p)$ must be far parentheses.

From the definition of a pioneer, there must exist exactly one pair of pioneers $(p', \mu(p'))$ such that $b(p') = b(p)$ and $b(\mu(p')) = b(\mu(p))$; and the pair $(p', \mu(p'))$ is the tightest enclosing pioneer pair of c. If there was a tighter enclosing pioneer pair, this pair would be enclosed by p and hence p would not be the tightest enclosing parenthesis. That the algorithm correctly computes p' is seen from the following:

(1) If c' is a closing parenthesis, then it must enclose c. It must be the tightest enclosing pioneer because it is the first pioneer to the right of c. Therefore p' = FINDOPEN(c').

(2) If c' is an opening parenthesis, then c and c' must share the same tightest enclosing pioneer parenthesis. Hence $p' = \text{ENCLOSE}(c')$.

Now, note that there are a number of (1 or more) far parentheses in $b(p)$ that have their matching parentheses in $b(\mu(p))$; the left-most of these far parentheses is p' and the rightmost is p. As has been observed before, there is only 1 pioneer in $b(p)$ that points to $b(\mu(p))$, and from the definition of a pioneer this means that there is no pioneer that occurs between p' and p.

Therefore, if q is the next pioneer in $b(p)$ to the right of p', then p must be the last far parenthesis in $b(p)$ before q, and if there are no pioneers to the right of p' in $b(p)$ then p must be the rightmost far parenthesis in the block. This is indeed what the above algorithm computes. We thus have:

Theorem 6. *A balanced string of* $2n$ *parentheses can be represented using* $2n + \mathrm{O}(n \lg\lg n / \lg n)$ *bits so that the operations* FINDOPEN, FINDCLOSE *and* ENCLOSE *can be supported in* $\mathrm{O}(1)$ *time.*

2.4. Simplifying the NND

Our structure, although conceptually simple, uses the (fairly complex) data structure of Theorem 1 as a subroutine. We now greatly simplify this subroutine as well, by modifying the definition of the pioneer family. Call a block *near* if it has no pioneer (and hence no far) parenthesis. We add to the pioneer family (as defined above) *pseudo*-pioneers consisting of the first and the last parenthesis of every near block (it is easy to see that the string corresponding to the modified pioneer family is balanced and has size $\mathrm{O}(N/\lg N)$).

We now argue that pseudo-pioneers do not affect the operations FINDOPEN, FINDCLOSE and ENCLOSE. For FINDOPEN(x) (FINDCLOSE(x)), where x is the position of a near parenthesis, the answer will be obtained by a table lookup. If x is the position of a far parenthesis, the answer is obtained by first searching for the previous (next) pioneer p. Since p will always be in $b(x)$, and $b(x)$ is not a near block, p cannot be a pseudo-pioneer and the earlier procedure goes through.

When we perform ENCLOSE(c) on an open parenthesis, where c is in a block that does not contain pseudo-pioneers, we first check to see if either the opening or the closing enclosing parenthesis is in the block using table lookup; if it is then we have computed ENCLOSE(c) correctly (with possibly one call to FINDOPEN). Otherwise, we locate the next pioneer c' after c and check to see if c' is an opening or closing parenthesis. It is possible that c' is a pseudo-pioneer that is an opening parenthesis, but if this is the case, the closest enclosing pioneer parenthesis pair of c is the same as that of c', and hence we get a valid result by performing ENCLOSE(p) on the pioneer bit-vector. If we wish to perform ENCLOSE(c) where c is in a near block and we cannot compute ENCLOSE(c) using table lookup (for example, if our block consists of pairs of opening and closing parentheses), then instead of computing ENCLOSE(c) we compute ENCLOSE(x) where x is the first parenthesis in the near block. We can still compute ENCLOSE(c) correctly using this method because the closest enclosing pioneer pair of c is the same, even with the pseudo-pioneers.

Since every block has at least a (pseudo-)pioneer, the gap between the positions of two successive pioneers in the modified pioneer family is at most $2B = \mathrm{O}(\lg N)$. This allows us to simplify the NND(s) in item (2) as follows.

2.4.1. A simple NND for uniformly sparse sets

We now consider the problem of creating an NND for a bit-vector of length M with 1s in a uniformly sparse set $S \subseteq [M]$ of positions. Specifically, we assume that $N = |S| = \mathrm{O}(M/\lg M)$ and further that if $S = \{x_1, \ldots, x_N\}$ and $x_1 < \cdots < x_N$, then for $i = 1, \ldots, N$, $x_i - x_{i-1} \leqslant (\lg M)^c$ for some constant $c \geqslant 1$ (take $x_0 = 0$). Our scheme uses four arrays of $\mathrm{O}(M \lg\lg M/\lg M)$ bits each and three tables of $\mathrm{O}(M^{2/3})$ bits each.

Let $t = \lfloor \lg M/(2c \lg\lg M) \rfloor$ and $S_t = \{x_t, x_{2t}, \ldots\}$. In the array A_1, we list the elements of S_t explicitly; i.e. for $i \geqslant 1$ we let $A[i] = x_{it}$. A_1 thus takes $\lceil N/t \rceil \cdot \lceil \lg M \rceil = \mathrm{O}(M \lg\lg M/\lg M)$ bits. In array A_2, we store the differences between consecutive elements of S, i.e., we let $A_2[i] = x_i - x_{i-1}$ for $i \geqslant 1$ (take $x_0 = 0$). Since all values in A_2 are $\mathrm{O}((\lg M)^c)$ by assumption, each entry can be stored using fixed-width entries of at most $z = \lceil c \lg\lg M \rceil$ bits each, and array A_2 takes $\mathrm{O}(N \lg\lg M)$ or $\mathrm{O}(M \lg\lg M/\lg M)$ bits in all. A table T_1 contains, for every bit string of length $tz \leqslant (\lg M)/2 + \mathrm{O}(\lg M/\lg\lg M)$ bits, the sum of the t values obtained by treating each group of consecutive z bits as the binary encoding of an integer. The table takes $\mathrm{O}(M^{2/3})$ bits.

Now SELECT(i, S) can be obtained in $\mathrm{O}(1)$ time as follows: let $i' = \lfloor i/t \rfloor$ and $i'' = (i+1) \bmod t$. Let $x = A_1[i']$ if $i' > 0$ and $x = 0$ otherwise. Obtain y as the concatenation of the values in $A_2[i'+1], A_2[i'+2], \ldots, A_2[i'+i'']$;

these $ti'' < tz$ bits are padded with trailing zeroes to make y be tz bits long (this is done in O(1) time by reading at most two $\lg M$-bit words from A_2 followed by masks and shifts), and we return $\textsc{select}(i, S) = x + T_1[y]$.

To support the RANK operation, we store two more arrays. We (conceptually) divide $[M]$ into blocks of t consecutive values, where the ith block b_i is $\{(i-1)t+1, \ldots, it\}$, and let $A_3[i] = |b_i \cap S_t|$, for $i = 1, \ldots, M/t$. Noting that $A_3[i] \in \{0, 1\}$, we conclude that A_3 takes $O(M/t) = O(M(\lg\lg M)/\lg M)$ bits. Viewing A_3 as the bit-vector of a set, the following standard auxiliary information permits O(1)-time RANK queries (details omitted) on A_3: an array A_4 containing, for $i = 1, \ldots, M/\lceil(\lg M)/2\rceil$, $A_3[1]+\cdots+A_3[i\cdot\lceil(\lg M)/2\rceil]$, and a table T_2 containing, for every bit-string of $\lceil(\lg M)/2\rceil$ bits, the number of 1s in that bit-string. Clearly A_4 occupies $O(M/t)$ bits and T_2 takes $O(\sqrt{M}\lg\lg M)$ bits.

Finally, we have another table T_3, which contains, for every bit string of length tz bits, interpreted as a sequence of t non-negative integers of z bits each, and a value $i \leqslant t(\lg M)^c$, the largest $l \leqslant t$ such that the sum of the first l of the t integers is less than i. As above, T_3 takes $O(M^{2/3})$ bits.

Now $\textsc{rank}(i, S)$ is implemented as follows. Let $i' = \lfloor i/t \rfloor$ and $r = \textsc{rank}(i', A_3)$. Observe that $A_2[r] = x_{rt}$ is the largest element in S_t that is $\leqslant i$. Let y be the concatenation of the values in $A_2[rt+1]$, $A_2[rt+2]$, $\ldots$, $A2[(r+1)t]$, and return $\textsc{rank}(i, S) = rt + T_3[y, i - x_{rt}]$. Thus we have:

Theorem 7. *Let $S \subseteq M$ be a subset of size $O(M/\lg M)$ and let the difference between two consective values of S be $O((\lg M)^c)$ for some constant c. Then there is a simple representation for S (using four arrays and three tables) taking $O(M\lg\lg M/\lg M) = o(M)$ bits in which the operations $\textsc{rank}(x, S)$ and $\textsc{select}(i, S)$ can be supported in constant time.*

Remark 8. Using Theorem 7 in place of Corollary 2, we get a parenthesis data structure that uses $2n + O(n\lg\lg n/\lg n) = 2n + o(n)$ bits, and is manifestly simple. Note that most applications involving succinct data structures (including the parenthesis one) would anyway require the table T_2.

Remark 9. The construction of this data structure is both simple and efficient: given a parenthesis string π of length $2n$, all auxiliary data structures can be constructed in $O(n/\lg n)$ time using additional $O(n\lg\lg n/\lg n)$ bits of workspace, as follows. We first determine the pioneer family of π. This is done in two passes over π, to determine the lists of closing and opening pioneers, respectively. By merging the two lists we produce the array A_2 of the NND. We determine the closing pioneers by processing each block in turn. We assume that when processing the ith block, there is a temporary stack that contains, for every block among the blocks $1, \ldots, i-1$ that has one or more currently unmatched (far) open parenthesis, the number of such parentheses. Clearly the space used by the stack is $O(n\lg\lg n/\lg n)$ bits. We use table lookup on the ith block to determine the number j of far closing parentheses in this block. If $j > 0$ then, using the stack and table lookup, it is easy to determine which of the far parentheses are pioneers. If there are no far parenthesis at all, we designate the last parenthesis as a (pseudo)-pioneer. Using table lookup, we determine the number of far open parentheses and push this on to the stack. If any closing pioneers are found we write their positions down in a temporary array A_c, again storing differences in positions of successive closing pioneers, rather than the positions themselves (since closing pioneers may be more than poly-log positions apart, A_c needs to be represented with a little care to fit in $O(n\lg\lg n/\lg n)$ bits). We calculate A_o, the positions of open pioneer parentheses, similarly, and merge A_o with A_c to give A_2. It is fairly easy to see that the entire process takes $O(n/\lg n)$ time.

3. Experimental evaluation

We now describe an experimental evaluation of this data structure. The aims are threefold—firstly, to evaluate the running time—more precisely, to determine the data structure's space–time trade-off in practice. It may be worth pointing out why there should be such a trade-off in the first place. Although our data structure has space $2n + o(n)$ bits, in a direct implementation of the theoretical version, the $o(n)$ term is greatly dominant for n even in the tens of millions. The space–time trade-off comes from making the lower-order term decay faster or slower with increasing n, and there are normally some very natural parameters that one can adjust to achieve this. For example, there is no need to limit the block size B to $\lceil(\lg n)/2\rceil$; one may choose B to be $c\lceil(\lg n)/2\rceil$, for some integer $c > 1$, and to operate on a block by means of c table lookups using *chunks* of $\lceil(\lg n)/2\rceil$ bits. This would tend to reduce space usage, while increasing the running time.

The second aim, which is standard methodology in experimental work, is to try and compare this data structure with another. A natural candidate is Jacobson's $O(n)$-bit data structure (modified for the RAM model) as it is quite simple. Jacobson did not compute the constant factors in the space usage, but it has been estimated that it uses $10n + o(n)$ bits [20]. Unfortunately, even for very small block sizes, the new data structure typically takes much less than $10n$ bits, making a comparison difficult. Thus, our second aim is to implement a variant of Jacobson's data structure that takes $(2 + \varepsilon)n + o(n)$ bits, for any constant $\varepsilon > 0$, in theory, with operations taking $O(1/\varepsilon)$ time, and use it for comparison.

Finally, for practical values of n, the lower-order terms in the space bound, and indeed the running time, are affected by a number of data-dependent parameters, such as the number of pioneer and far parentheses. Thus, the third aim is to study the data-dependent parameters, both on random trees and trees derived from real-life XML files.

3.1. Jacobson's data structure

We now describe our variant of Jacobson's data structure, which comprises the parenthesis string, divided into blocks of B bits each, together with four auxiliary data structures:

- An array M_o, such that $M_o[i] = \mu(p)$, if p is the ith opening pioneer. An array M_c stores the analogous information for closing pioneers. By Lemma 3, there are at most $4 \cdot (2n/B)$ entries of $\lg n$ bits each in both arrays combined. This takes at most εn bits, for any constant $\varepsilon > 0$, by choosing $B = c\lceil(\lg n)/2\rceil$ for a sufficiently large constant c, and operating on a block using c table lookup operations.
- A set $S_o \subseteq \{1, \ldots, 2n\}$ that contains the indices of parentheses corresponding to opening pioneers, and an analogous set S_c for closing pioneers. By implementing an NND on S_o and S_c, we can index into M_o and M_c. Since S_o and S_c have size $O(n/\lg n)$, the NNDs take $o(n)$ bits, by Corollary 2. (Jacobson used $2n + o(n)$ bits for each of S_o and S_c.)
- An array E_l such that $E_l[i]$ stores the left excess at the first parenthesis position of block i; and an array E_r such that $E_r[i]$ stores the right excess at the last parenthesis position of block i. E_l and E_r can be stored using $o(n)$ bits by storing the numbers of opening parenthesis to the left of each block; this sequence of numbers can be stored in $o(n)$ bits using Corollary 2 and ideas from (e.g.) [21].
- An array D, which for every block b, such that the left excess at the first parenthesis of b is $e > 0$, stores the position of the first parenthesis to the left of b (counting from the start of b) with excess $e - 1$ (a null value is stored if $e = 0$). Clearly, by selecting B appropriately, D also takes at most εn bits, for any constant $\varepsilon > 0$.

As in the new parenthesis data structure, we use a constant number of tables, occupying at most $o(n)$ bits, that allow us to operate on chunks, and hence blocks, in constant time. We see that the space used, in total, is $(2 + \varepsilon)n + o(n)$ bits, for any constant $\varepsilon > 0$.

We now show how to implement FINDCLOSE. Let p be an open parenthesis at position i. Then table lookup gives the position of $\mu(p)$ inside its block or determines that it is far. If p is far, then $b(\mu(p)) = b(\mu(p^*))$, where p^* is the previous pioneer (RANK1 and SELECT1 on S_o gives p^* and the array M_o gives $\mu(p^*)$). We can find the left excess e between p^* and p by computing the left excess at p and p^* by two table lookups and using the array E_l. Now $\mu(p)$ is the first closing bracket in that $b(\mu(p^*))$, counting from the left, with right excess i, which can be found by table lookup.

Jacobson did not support ENCLOSE(c) operation. Let c be the position of an open parenthesis. As noted in [20], if the left excess e at c is 0 (found from array E_l and table lookup), then there is no enclosing parenthesis. Otherwise, the left parenthesis of the answer is the previous parenthesis with left excess $e - 1$. This can be found by table lookup and array D. All operations take $O(1/\varepsilon) = O(1)$ time.

3.2. Implementations

An important difference between the implementations and theoretical descriptions is that we do not use chunks of size $\lceil(\lg n)/2\rceil$. In practice, the chunk size is constrained by the need to have all tables fit into cache memory in the computer, and to very efficiently extract a chunk from a parenthesis string. For this reason, we choose chunk sizes of 8 or 16 bits, and block sizes $B = 32, 64, 128$ or 256. There are a number of other important variations from the theoretical data structures, which we now describe. We also calculate the space requirements of the data structures, excluding the space for the tables.

3.2.1. Implementation of the new NND

Let $S \subseteq [M]$ be the set to be stored. We assume that the gaps between successive values are no more than 256, allowing A_2 to consist of 8-bit values. Each element of array A_1 is a 32-bit value. Array A_4 is implemented by storing with each block of B bits in A_3 a 32-bit value for the number of 1s to the start of the block.

In order to reduce the space requirements for arrays A_1, A_3 and A_4 we would like t to be large; however, for practical values of M, this would lead to impracticably large tables. In our implementation we use values of $t = 2, 4, 8$ and 16, but abandon the tables T_1 and T_3, replacing each operation on T_1 and T_3 by upto t operations. The space used by the NND implementation (in bits) is:

$$f(|S|, M) = \frac{32|S|}{t} + 8|S| + \frac{M}{t} + \frac{32M}{tB}. \tag{1}$$

3.2.2. Implementation of the new $2n + o(n)$ parenthesis DS

For simplicity (and to make it easier to share tables), we use the same block and chunk sizes at each level of recursion, and also in the NND. In the recursive data structure we have $2n$ parentheses and p_1 pioneers at the top level, p_1 parentheses and p_2 pioneers at the next level. Each of these levels stores the parentheses and an NND for the pioneer positions. From Eq. (1), the number of bits required for these two levels is

$$\begin{aligned} &2n + 2p_1 + f(p_1, 2n) + f(p_2, p_1) \\ &\quad = 2n + 2p_1 + \frac{32p_1}{t} + 8p_1 + \frac{2n}{t} + \frac{64n}{tB} + \frac{32p_2}{t} + 8p_2 + \frac{p_1}{t} + \frac{32p_1}{tB}. \end{aligned}$$

At the bottom of the recursion, we store, with each parenthesis p, 32-bit addresses for each of: the match of p and the enclosing open parenthesis for p. This adds up to $64p_2$ bits.

For our experiments we need to support RANK on the top level parenthesis string. We first divide the $2n$ parentheses into *superblocks* of size 2^{16}. We store with each superblock the number of 1s to its left (using 32-bit values) and with each block we store the number of 1s from the start of its superblock (using 16-bit values). The number of bits required for supporting RANK is therefore $64n/2^{16} + 32/B$ bits.

Summing up and simplifying, the total number of bits required for the new parenthesis data structure, augmented to also support RANK on the top level parenthesis string is

$$\left(2.001 + \frac{64}{tB} + \frac{32}{B} + \frac{2}{t}\right)n + \left(9 + \frac{33}{t} + \frac{32}{tB}\right)p_1 + \left(72 + \frac{32}{t}\right)p_2. \tag{2}$$

Note that the NND's limit of 256 on the gap size means that $B \leqslant 128$ (even after pseudo-pioneers are inserted, two pioneers could be $2B - 2$ positions apart), whereas t could, in principle, be arbitrarily large. Thus, the best space usage of this implementation is $(2.96 + O(1/t))n$ bits, if p_1 and p_2 are as large as their theoretical maxima.

3.2.3. Implementation of the modified Jacobson data structure

We have two 32-bit arrays similar to M_o and M_c, with the difference that if p is the ith pioneer, then $M_o[i]$ contains a 32-bit value, whose lowest 8 bits store p's position within its block (this means $B \leqslant 256$), and the top 24 bits store $b(\mu(p))$ (our experience was that it did not help to precisely locate $\mu(p)$).

We do not compress the E_l and E_r, instead we effectively store just the number of open parentheses before each block (rank information), from which E_l and E_r can be readily calculated. This rank information takes 32 bits per block. The array D has 32-bit values.

The sets S_o and S_c are represented as follows. With each block we store two 8-bit numbers that give the numbers of open and close pioneers in each block. Every four blocks, we store two 32-bit numbers that record the prefix sums of these 8-bit values. This information allows us to localise the segments of M_o and M_c that correspond to a given block, and searching the appropriate segment allowed us to find the nearest pioneer (from a theoretical perspective one could view this as potentially a $o(n)$-bit but $O(\lg B)$ time solution).

To summarise, in addition to the $2n$ bit string, the space used per block is $(80 + (32 + 32)/4) = 96$ bits. Since there are $2n/B$ blocks, the space is $192n/B$ bits. In addition, the space for M_o and M_c is 32 bits times their size. Thus, the space used is

$$2n + 192n/B + 32 \cdot (|M_o| + |M_c|). \tag{3}$$

Basic data on XML files (1 of 2)					
				Slowdown	
File	size	nodes	% Far	UltraSparc-III	Pentium 4
dblp.xml	1.34E+08	9995781	5.632	1.268	
desc2004.xml	2.39E+08	16200984	7.353	1.321	
elts.xml	116505	5991	4.791	2.469	2.313
lineitem.xml	32295475	2045953	5.882	1.280	2.303
mondial-3.0.xml	1784825	57372	5.398	2.450	1.907
nasa.xml	25050288	1425535	6.683	1.269	2.419
orders.xml	5378845	300003	6.250	1.461	2.249
partsupp.xml	2241868	96003	6.251	2.578	1.847
pcc1.xml	48750	3562	19.652	2.815	4.558
pcc2.xml	252689	17857	18.699	2.804	4.175
pcc3.xml	179638	13051	18.412	2.868	3.943
play1.xml	260951	18967	5.357	2.556	2.439
play2.xml	141336	9423	5.264	2.564	2.556
play3.xml	288814	19840	5.071	2.505	2.328
sprot.xml	10579	805	4.845	3.131	2.220

Fig. 2. Test file names, file sizes, XML nodes, % far parenthesis, slowdown relative to DOM for a DFS traversal on Sun UltraSparc-III and Pentium 4, when $B = 64$ and using 16-bit chunks.

As in the new parenthesis data structure, the space requirements for our version of Jacobson's data structure depend on the number of pioneer parentheses. Assuming a maximum value of $B = 256$, and maximum values for the numbers of pioneers, we get a minimum worst-case space bound of about $3.75n$ bits.

3.3. Experimental results

3.3.1. XML statistics

In order for us to be able to understand the space requirements of our representations in real-life situations, we gathered statistics from 33 real-world XML files [23,25]. The files come from different applications and have different characteristics. Figs. 2 and 3 give the basic statistics of these files such as sizes and number of nodes.

Two key parameters that would affect the space and running time of the data structure are the proportion of far parentheses, and the number of pioneers, for a given block size. The former is important, as the computation time for many operations is higher for a far parenthesis. The proportion of far parentheses, in parenthesis strings derived from XML data, is shown in Fig. 4(b). We note that even though it is possible to have essentially $2n$ far parentheses, the observed proportion of far parentheses is very small, and decreases with increasing blocksize. The average value hides some (fairly large) variations: for example, a family of files called `pcc1.xml`, `pcc2.xml` and `pcc3.xml` have up to 27% far parentheses (for a block size of 32). Each of these files contains a proof of correctness and is highly nested.

Secondly, the space usage of the data structure depends on the number of pioneers. Note that it is somewhat uninformative to plot the fraction of pioneer parentheses against block size B. As the worst-case number of pioneers is at most $4n/B$, as the block size increases, the proportion of pioneers drops. Instead, we plot the number of pioneers *per block* versus the block size. In Fig. 4(a) we show the number of pioneers per block using Jacobson's definition, the pioneer family definition and also for the set including pseudo-pioneers. Note that the theoretical worst-case bound for the number of pioneers per block is essentially 4, but on average there were less than 2.4 pioneers per block, or 60% of

Basic data on XML files (2 of 2)					
				Slowdown	
File	size	nodes	% Far	UltraSparc-III	Pentium 4
stats1.xml	671949	56448	5.104	2.627	2.237
stats2.xml	617523	49500	5.584	2.697	2.322
supp2004.xml	4.13E+08	30322317	7.435	1.286	
tal1.xml	734541	49876	10.715	2.654	3.736
tal2.xml	510060	34660	10.462	2.676	3.735
tal3.xml	251669	16821	11.432	2.659	3.767
tpc.xml	300548	35290	4.384	2.454	2.095
treebank.xml	6680	798	10.025	3.217	2.791
treebank_e.xml	86082517	7312612	10.910	1.420	
votable.xml	15908196	1991192	5.483	1.260	2.326
w3c1.xml	220869	10094	8.064	2.709	2.431
w3c2.xml	196308	9090	7.492	2.657	2.432
w3c3.xml	201918	7778	7.393	2.646	2.427
w3c4.xml	105011	4519	7.480	2.714	2.713
w3c5.xml	247538	8422	6.412	2.551	2.246
weblog.xml	2295	135	6.667	3.530	2.519
XCDNA.xml	6.08E+08	25221153	5.553	1.245	
XPATH.xml	52246714	2522571	5.780	1.245	2.281

Fig. 3. Test file names, file sizes, XML nodes, % far parenthesis, slowdown relative to DOM for a DFS traversal on Sun UltraSparc-III and Pentium 4, when $B = 64$ and using 16-bit chunks.

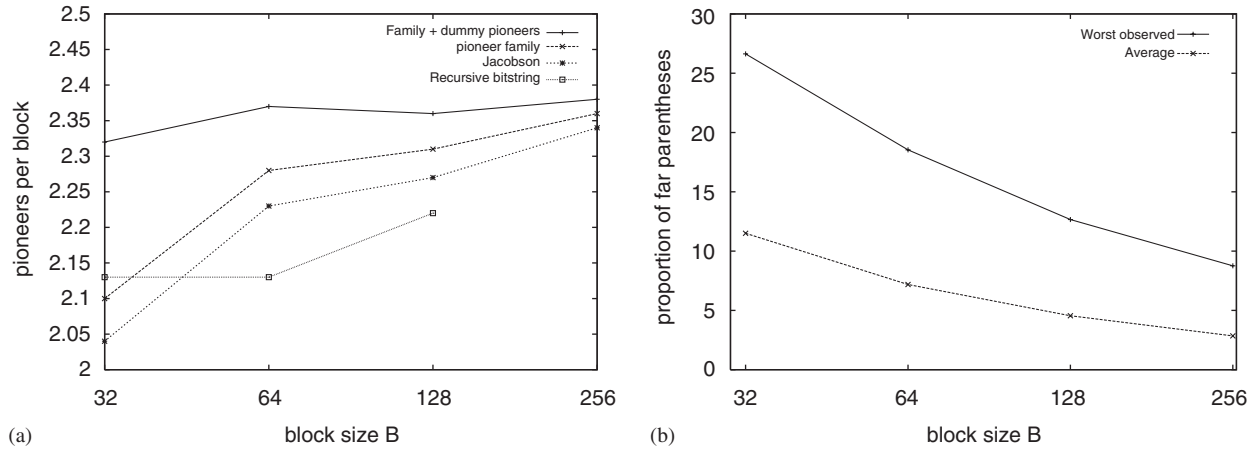


Fig. 4. Statistics on XML files. The x-axis has the block size B and the y-axis has: (a) number of pioneers per block, for all three definitions of pioneers, together with the number of pioneers per block in the recursive parenthesis string and (b) the proportion of parentheses that are far.

the theoretical maximum number of pioneers; this held regardless of which definition was used. The same holds for the pioneers in the parenthesis sequence in the first level of recursion; in fact the proportion of pioneers is possibly a little less there. Again, the files `pcci.xml` are well above average, and have over 3.7 pioneers per block for some values of B.

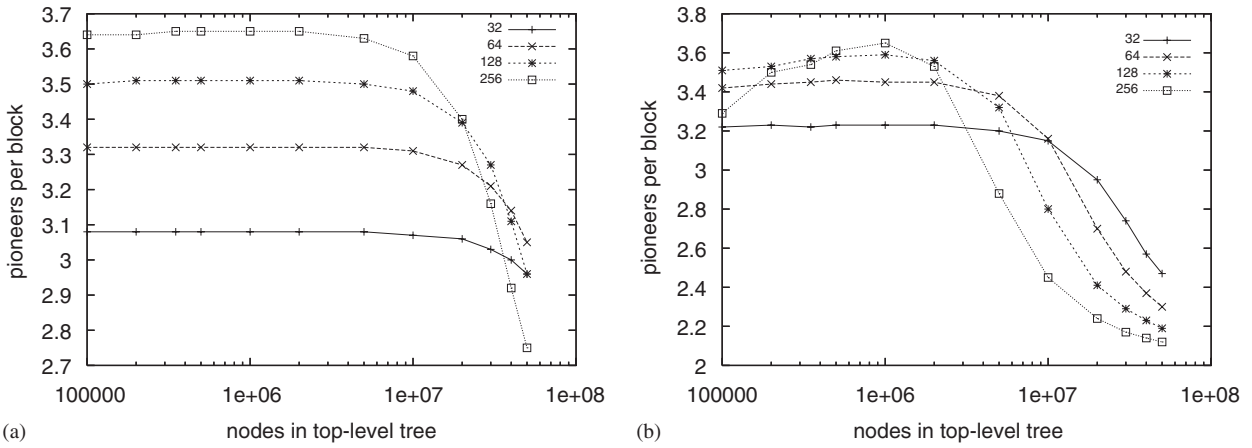


Fig. 5. Statistics on random trees. The x-axis has the number of nodes in the original tree and the y-axis has the average number of pioneers per block over 50 sequences for the first 6 data points and over 20 sequences for the next 6 data points (using the pioneer family plus pseudo-pioneers) in (a) the original string and (b) in the recursive parenthesis string.

	$PD = 4$		$PD = 2.4$	
Blocksize	Jacob	New	Jacob	New
32	16.00	8.34	12.80	5.75
64	9.00	4.65	7.40	3.73
128	5.50	3.24	4.70	2.86
256	3.75		3.35	

Fig. 6. Space used by the modified Jacobson implementation and new data structure, excluding tables, assuming a pioneer density of 2.4 per block and 4 per block, respectively, and taking $t = 16$ in the new data structure. The units are bits per parenthesis pair, i.e. the space, in bits, used to represent $2n$ parenthesis by a data structure using a particular value of B is obtained by multiplying the corresponding entry by n.

We have also obtained statistics for random ordinal trees (or random sequences of balanced parentheses), generated using software from [16]. We took balanced parentheses strings of a number of lengths, starting from 200,000 to 100,000,000. For each length, we generated between 20 and 50 random balanced strings and computed the following statistics: number of far parenthesis and the size of the modified pioneer family. We did this both for the original random parenthesis string and the parenthesis string comprising the modified pioneer family. As can be seen from Fig. 5(a), for moderately sized sequences of up to 10 million nodes, the pioneer density in the original bitstring is high (varying between 3.07 and 3.64 per block, depending on blocksize), but appears to drop off for larger sequences. In Fig. 5(b), the x-axis value is the size of the *original* parenthesis sequence; thus, the corresponding y-values are really for significantly shorter parenthesis sequences, and the drop-off is even faster than it appears, relative to that of Fig. 5(a). Although not shown here, the proportion of far parentheses shows different behaviour, starting fairly high and gradually rising (e.g., for $B = 128$, it is about 13.7% for sequences of length 200,000, and rises gradually to about 17% for sequences of length 100,000,000). An analytical treatment of this phenomenon would be interesting.

We calculate the space bounds (excluding tables) used by our data structure, taking both a worst-case pioneer density of 4 per block and an average-case density of 2.4 per block (an upper bound on the average values obtained from XML data). We use Eqs. (2) and (3) to calculate these values and the results are shown in Fig. 6.

3.3.2. Performance evaluation

We implemented the data structures in C++, and ran some tests on a Pentium 4 machine and a Sun UltraSparc-III machine. The Pentium 4 has 512 MB RAM, a 2.4 GHz CPU and a 512 KB cache, running Debian Linux. The compiler was `g++` 2.95 with optimisation level 2. The Sun UltraSparc-III has 8 GB RAM, a 1.2 GHz CPU and a 8 MB cache, running SunOS 5.9. The compiler was `g++` 3.3 with optimisation level 2. The code of both the modified Jacobson's and the new data structure was highly optimised, using standard tricks such as unrolling large loops, using inline functions, etc.

In addition, we optimised the cache performance of the data structures. For example, we have clustered data in the NND to improve locality by placing an element of array A_1 contiguously in memory with t elements of A_2, since these are accessed together during a SELECT operation. We also place a block of B elements from A_3 near the prefix sum of the 1s in all preceding blocks. We have also optimised our accesses to two-dimensional (2-D) tables. An example of such tables is one that is indexed by a 16-bit chunk of parentheses, together with an excess value (up to 16) and contains the location of the leftmost open parenthesis with the given excess value. Assuming that each entry of a table is a byte, the table in the above example takes $2^{16} \cdot 16$ bytes or 1 MB, exceeding the cache size on Pentium 4. We minimise accesses to such tables by using them strictly when needed. Such 2-D tables are used to find a matching opening or closing parenthesis in a block. This involves skipping over several chunks which do not contain the matching parenthesis—we use 1-D tables to skip over these chunks and use a 2D table only once we have identified the chunk that contains the matching parenthesis. A further optimisation when using 16-bit chunks is to replace the single access to a two-dimensional 16-bit table with 2 accesses to a two-dimensional 8-bit table. A result of these optimisations is that both on the Pentium 4 and the UltraSparc, 16-bit chunks were typically slightly faster than 8-bit chunks.

Running times were evaluated against a baseline of CenterPoint XML's DOM implementation, release 2.1.7. In this implementation, each node stores pointers only to its parent, first child and next sibling. This means that each node requires $3 \cdot 32 = 96$ bits; however, operations such as `getLastChild` or `getPreviousSibling` require traversal over all child nodes or many siblings. The test in each case was to traverse the tree in DFS and in BFS order and count the number of nodes of a particular type. The DFS traversal in DOM uses only the methods `firstChild` and `nextSibling`, using the recursion stack to go back to the parent. Each method call essentially involves only following a pointer. To emulate this in the parenthesis data structure, we identify a node by the position of its open parenthesis. Then, emulating `firstChild` is trivial, but emulating `nextSibling` requires a call to FINDCLOSE. Both BFS algorithms used the C++ Standard Template Library queue implementation. We performed multiple tree walks in order to get more stable running times; repetitions varied from 10 (for the large files) to 500,000 (for the smaller files). To keep the tests fair, we ensured that files fit in memory, even using DOM. This meant that we did not run tests using `XCDNA.xml`, `dblp.xml`, `desc2004.xml`, `supp2004.xml` and `treebank_e.xml` on the Pentium 4.

Figs. 7 and 8 summarise the running times. Since we used real-world data, each data point in these charts is derived from a relatively small number of files, which have widely differing characteristics. Not unexpectedly, there is a lot of variation around the average. Nevertheless, certain broad trends are clear. The average slowdown increases with blocksize, this is because we need to process more chunks per block. In general, for a DFS traversal, on the Pentium 4 machine the data structures were 1.7 to 4 times slower than the DOM implementation. On the UltraSparc-III the data structures were in general 1 to 2.5 times slower. The DOM implementation does mainly memory accesses whereas the parenthesis data structures do a lot of computation. The difference in the slowdown between the UltraSparc-III and the Pentium 4 machines is mainly due to the different machine architectures and the cost of memory accesses. It is also due to the fact that on the UltraSparc-III the parenthesis data structures and tables reside in cache memory.

The average slowdown on both systems was less for a BFS traversal. See Figs. 7 and 8. This is very good considering that DOM simply follows pointers, and given the limited connectivity of this DOM implementation, the gap could have been much smaller for a general traversal (e.g. visiting nodes in reverse DFS order).

The key benefit is that the space to store the tree structure is drastically reduced. For example, with $B = 128$ we can choose between $4.7n$ bits and $2.86n$ bits, on average, to represent the tree structure, while the DOM implementation takes 96 bits and suffers an average slowdown between 2.58 and 2.87 on a Pentium 4 for a DFS traversal. Of course, we do not expect a reduction by a factor of 30 to 35 in the overall size of an in-memory XML representation but pointers are a considerable part of such representations. The gap between the new DS and the modified Jacobson is fairly narrow. For the same value of B, the new DS uses a lot less space, but is slower. Fig. 9 shows the trade-off between space usage and speed. The data points are derived from the average slowdown over all files and the bits per node assuming

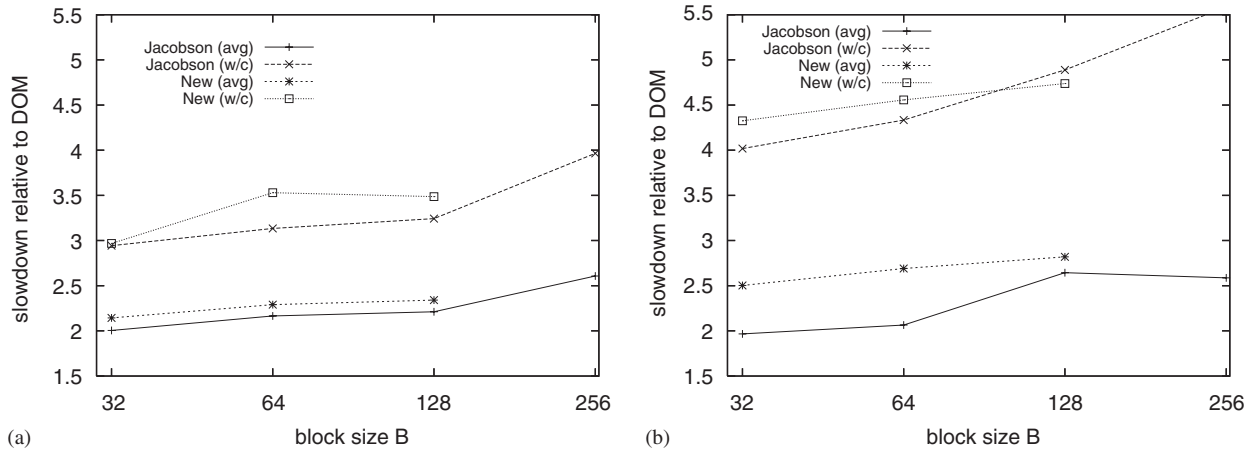


Fig. 7. DFS tree traversal: Average slowdown relative to DOM over all files and worst instance over all files. The x-axis has the block size B and the y-axis has slowdown relative to DOM: (a) on a Sun UltraSparc-III, (b) on a Pentium 4.

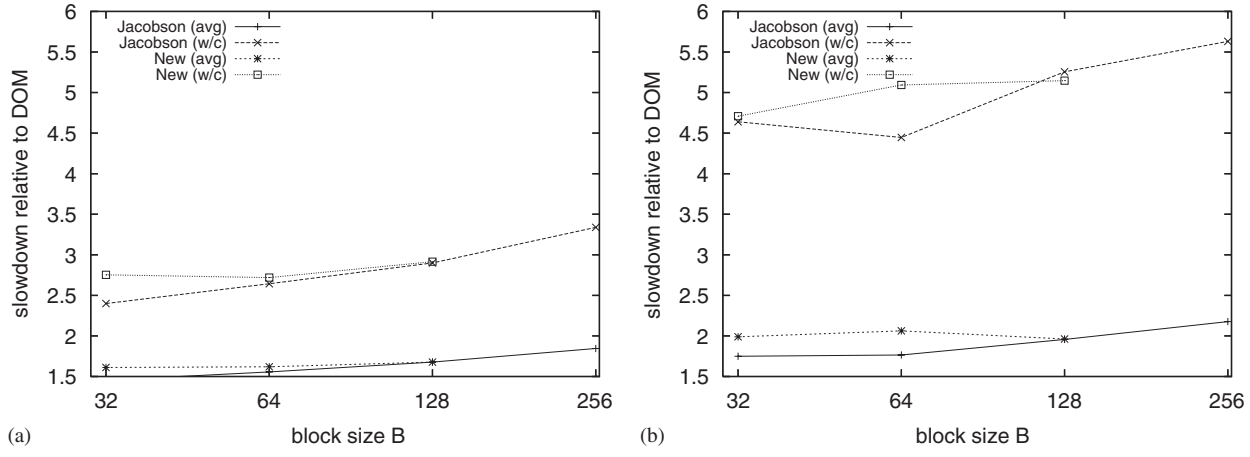


Fig. 8. BFS tree traversal: Average slowdown relative to DOM over all files and worst instance over all files. The x-axis has the block size B and the y-axis has slowdown relative to DOM: (a) on a Sun UltraSparc-III, (b) on a Pentium 4.

a pioneer density of 2.4 for Jacobson at $B = 32, 64, 128$ and $B = 256$ and for the new data structure at $B = 32, 64$ and $B = 128$. This shows that on the Ultra Sparc-III the new data structure offers a better trade-off between speed and time, while on the Pentium-III the modified Jacobson appears slightly better.

Analysing the performance further, Fig. 10 shows the cache misses per node for a DFS traversal of some of our data files on the Pentium 4 when using our new recursive parenthesis data structure. We have removed files which have very similar structures. These values were obtained using `Cachegrind` [24]. As can be seen, the number of cache misses is very low; this is partly due to the inherent locality in a DFS traversal of a parenthesis structure, but also reflects our optimisations. (Note that for many of these files the entire data structure fits in cache.)

One may further theorise that the running time is dependent on the proportion of far parentheses, on the grounds that the FINDCLOSE computation for near parentheses is simple, involving essentially only table lookup and local access to the parenthesis string. Fig. 11 shows for our new data structure slowdown for a DFS traversal versus percentage of far parentheses on the Pentium 4 and a blocksize of 64 bits. On the Pentium 4 slowdown tends to increase with the proportion of far parentheses, while on the UltraSparc-III the effect is significantly more muted.

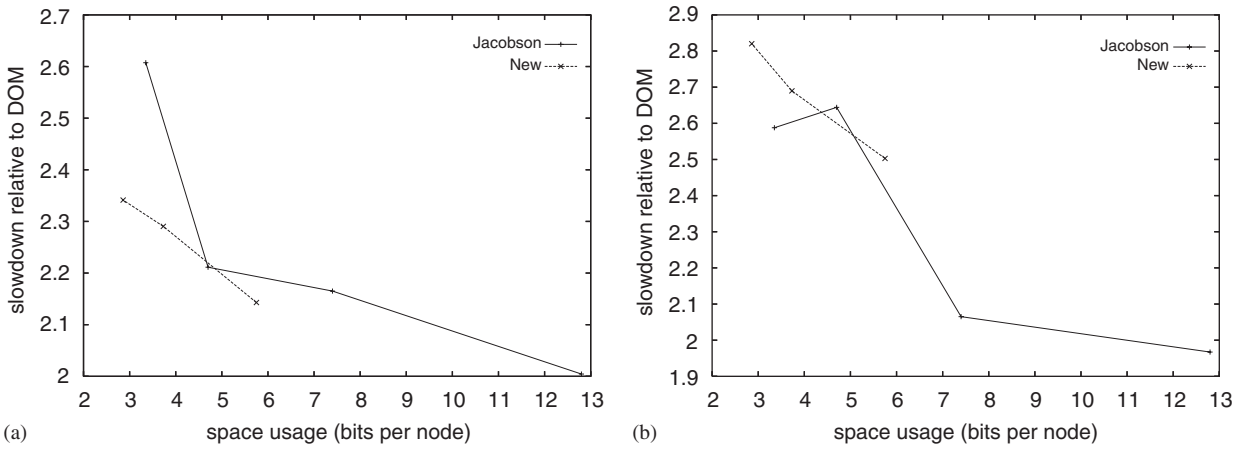


Fig. 9. Comparison of space usage and speed. The x-axis has the space usage, in terms of bits per tree node (or parenthesis pair) and the y-axis has average slowdown relative to DOM: (a) on a Sun UltraSparc-III, (b) on a Pentium 4.

	$B = 32$		$B = 64$		$B = 128$	
	8	16	8	16	8	16
Average	0.014	0.051	0.010	0.041	0.009	0.040
Maximum	0.062	0.140	0.045	0.111	0.036	0.124

Fig. 10. Average and maximum cache misses per node on Pentium 4 for the new recursive data structure during a DFS traversal, when $B = 32, 64, 128$ using 8-bit and 16-bit chunks. In all cases the minimum value was 0.

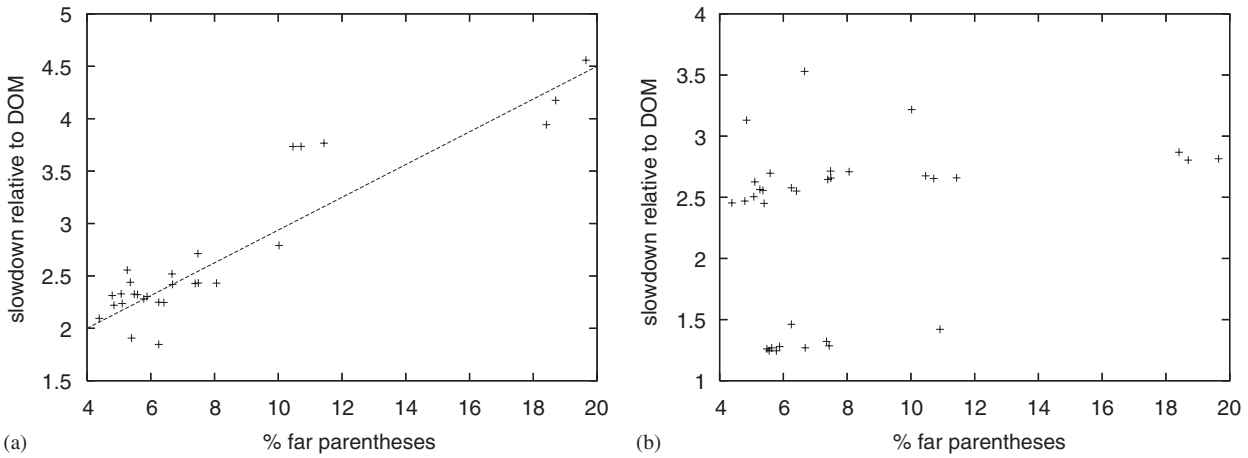


Fig. 11. Slowdown versus percentage of far parentheses using 64-bit blocks: (a) on a Pentium 4 and, (b) on a Sun UltraSparc-III. Each point represents data for one of our sample XML files.

4. Conclusions and further work

We have given a conceptually simple succinct representation for balanced parentheses that supports natural parenthesis operations in constant time. This immediately gives a simpler optimal representation for all applications of these

data structures. The new representation has theoretical advantages as well, such as a simple sublinear-time and space construction algorithm, and an improved lower-order term in the space bound.

A number of questions arise from the experimental data. It would be interesting to obtain analytical results regarding the number of pioneers or far parentheses in random trees, as well as an accurate cache analysis of the parenthesis data structure. Possibly the data structure can be further simplified.

Acknowledgement

We would like to thank R. Ramani for coding versions of the rank/select data structures and D. Okanohara for helpful comments.

References

[1] Apache Xindice Project, Frequently asked Questions 10, ⟨http://xml.apache.org/xindice/faq.html#faq-N10084⟩.
[2] D.A. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, TR 2001/46, Department of Maths and Computer Science, University of Leicester, 2001.
[3] D.A. Benoit, E.D. Demaine, J.I. Munro, V. Raman, Representing trees of higher degree, in: Proc. Sixth WADS, Lecture Notes in Computer Science, Vol. 1663, Springer, Berlin, 1999, pp. 169–180.
[4] Centerpoint XML, ⟨http://www.cpointc.com/XML⟩.
[6] K. Chupa. MMath Thesis, University of Waterloo, 1997.
[7] D. Clark, J.I. Munro, Efficient suffix trees on secondary storage, in: Proc. Seventh ACM-SIAM SODA, 1996, pp. 383–391.
[8] J.J. Darragh, J.G. Cleary, I.H. Witten, Bonsai: a compact representation of trees, Software-Practice and Experience, Vol. 23, 1993, pp. 277–291.
[9] P. Ferragina, G. Manzini, An experimental study of a compressed index, Inform. Sci. 135 (2001) 13–28.
[10] M.L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with O(1) worst case access time, J. ACM 31 (1984) 538–544.
[11] R.F. Geary, R. Raman, V. Raman, Succinct ordinal trees with level-ancestor queries, in: Proc. 15th ACM-SIAM SODA, 2004, pp. 1–10.
[12] R. Grossi, A. Gupta, J.S. Vitter, When indexing equals compression: experiments on suffix arrays and trees, in: Proc. 15th ACM-SIAM SODA, 2004, pp. 629–638.
[13] T. Hagerup, P.B. Miltersen, R. Pagh, Deterministic dictionaries, J. Algorithms 41 (1) (2001) 69–85.
[14] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th FOCS, 1989, pp. 549–554.
[15] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C Recommendation 13 November, 2000, ⟨http://www.w3.org/TR/DOM-Level-2-Core⟩, W3C Consortium, 2000.
[16] H.W. Martin, B.J. Orr, A random binary tree generator, in: Computer Trends in the 1990s, Proc. 1989 ACM 17th Annu. Computer Science Conference, ACM Press, New York, 1989, pp. 33–38.
[17] I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representation of permutations, in: Proc. 30th ICALP, Lecture Notes in Computer Science, Vol. 2719, Springer, Berlin, 2003, pp. 345–356.
[18] I. Munro, V. Raman, S.S. Rao, Space efficient suffix trees, J. Algorithms 39 (2001) 205–222.
[19] J.I. Munro, Tables, in: Proc. 16th FST&TCS Conference, Lecture Notes in Computer Science, Vol. 1180, Springer, Berlin, 1996, pp. 37–42.
[20] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Computing 31 (2001) 762–776.
[21] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th ACM-SIAM SODA, 2002, pp. 233–242.
[22] K. Sadakane, Succinct representations of lcp information and improvements in the compressed suffix arrays, in: Proc. 13th ACM-SIAM SODA, 2002, pp. 225–232.
[23] University of Washington XML Repository, ⟨http://www.cs.washington.edu/research/xmldatasets/⟩.
[24] Valgrind—A GPL'd system for debugging and profiling x86-Linux programs, ⟨http://valgrind.kde.org/⟩.
[25] VOTable Documentation, ⟨http://www.us-vo.org/VOTable/⟩.