Dynamizing Succinct Tree Representations

Stelios Joannou and Rajeev Raman

University of Leicester, Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH.

Abstract. We consider *succinct*, or space-efficient, representations of *ordinal* trees. Representations exist that take $2n + o(n)$ bits to represent a *static* n-node ordinal tree – close to the information-theoretic minimum – and support navigational operations in $O(1)$ time on a RAM model; and some implementations have good practical performance.
The situation is different for *dynamic* ordinal trees. Although there is theoretical work on succinct dynamic ordinal trees, there is little work on the practical performance of these data structures. Motivated by applications to representing XML documents, in this paper, we report on a preliminary study on dynamic succinct data structures. Our implementation is based on representing the tree structure as a sequence of balanced parentheses, with navigation done using the *min-max* tree of Sadakane and Navarro (SODA '10). Our implementation shows promising performance for update and navigation, and our findings highlight two issues that we believe will be important to future implementations: the difference between the *finger model* of (say) Farzan and Munro (ICALP '09) and the *parenthesis model* of Sadakane and Navarro, and the choice of the balanced tree used to represent the min-max tree.

1 Introduction

A number of applications that involve indexing and processing textual or semi-structured data now need to deal with increasingly large volumes of data. Typically, these applications do not have satisfactory external-memory solutions and so the data has to be held and processed in main memory; examples include the various applications of suffix trees and a number of XML processing tasks (including XQuery search, XSLT transformation). An important bottleneck in many such applications is the space required to represent some kind of tree-structured object: in such applications, *succinct*, or highly space-efficient, representations of trees[18] are having increasing impact. The focus of this paper is on *ordinal trees*, which are arbitrary rooted trees where the children of each node are ordered. As there are $\frac{1}{n}\binom{2n-2}{n-1}$ ordinal trees on n nodes, storing an ordinal tree requires $2n - O(\lg n)$ bits, as opposed to the standard representation that takes asymptotically $\Theta(n)$ words, or $\Theta(n \lg n)$ bits, of memory. In recent years, a number of representations of *static* ordinal trees have been developed [14, 15, 5, 13, 20] that use $2n + o(n)$ bits of memory, and support a wide range of navigational operations in $O(1)$ time assuming the RAM model of computation with word size $\Theta(\log n)$ (the default theoretical model that we will assume in this

paper). The excellent practical performance of succinct ordinal tree representations has been shown in many papers including [8, 12, 2]. For example, when representing XML documents, which are essentially ordinal trees, a standard pointer-based representation [1] has *five* pointers per node[1] (320 bits per node on a 64-bit machine) to represent the tree structure and support fast navigation; thus, the attractiveness of a representation that takes just a few bits per node but supports operations quickly in practice, is clear. Indeed, succinct ordinal trees have been successfully applied to several XML applications [25, 9, 3, 7].

Despite the success of static succinct data structures, more needs to be done. For example, in the XML context, efficient support for updates to documents is fundamental: the W3C standard DOM API specifies a number of methods for modifying XML documents [24]. In discussions with industry contacts we have found that there are few "purely static" real-world XML applications that deal with large documents. While static succinct trees have received a lot of attention, there has been much less work on dynamizing them, in theory or in practice. Farzan and Munro [10] and Sadakane and Navarro [20] studied the theory of dynamic succinct ordinal trees. Practical studies of dynamic succinct data structures are few, and we are only aware of one work that discusses the implementation of a dynamic succinct ordinal tree [25]; however, they implement a theoretically non-optimal tree, and also their (good) perfomance results are for their entire system, rather than the tree in isolation.

In this paper, we consider practical performance issues in implementing dynamic succinct trees. The operations we consider are:

- Basic navigation: `first-child`, `last-child`, `parent`, `next-sibling`, `prev-sibling`.
- Updates: insertion and deletions of leaves.

Furthermore, we are concerned not just with the performance of individual operations, but also focus on *traversals*, or relatively long sequences of navigational operations. We give a brief introduction to the approach we take to dynamization, before summarizing our main contributions.

Our approach is to represent the current n-node ordinal tree as a *balanced parenthesis (BP)* sequence of length $2n$ (see Fig. 1). For specificity, assume that a node is represented by the opening parenthesis of the pair representing it. We obtain the BP sequence representing a tree by going through a tree depth-first, outputting an opening parenthesis when a node is first encountered and a closing one when every node of its sub-tree has been encountered. The BP sequence is divided into *blocks* of size $\Theta(B)$ for some parameter B–in theory $B = \Theta((\log n)^2)$–and a *min-max* excess tree, a balanced binary tree, is stored over the blocks to perform *excess search* [17, 20] (see Section 2 and 4 for details). So far, we are following the approach proposed by [20], who show that if some details are handled carefully, this approach yields an implementation with space bound $2n + o(n)$ bits, and time bounds $O(\log n)$ for both navigation and update operations; the time bound can further be reduced to $O(\log n / \log \log n)$ while

[1] Parent, first child, last child, previous sibling and next sibling.

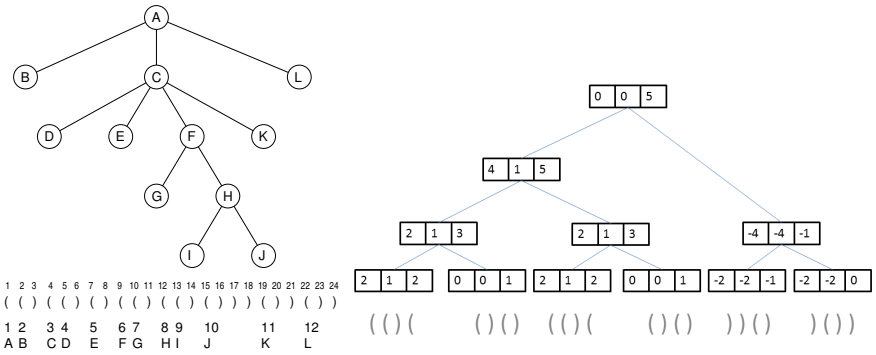

Fig. 1. Example ordinal tree and its BP representation (left), the min-max tree (right).

maintaining the space bound. However, more work needs to be done to obtain a satisfactory practical data structure from this idea, as we now discuss.

Parentheses versus Fingers. The first question that arises is what the precise interface through which the data structure implements the navigation operations should be. Navigation in the BP is usually understood [15] in terms of the following two operations:

- `findclose`(i): if the i-th parenthesis is an opening parenthesis, then find the position of the matching closing parenthesis (`findopen` is similar).
- `enclose`(i): find the opening parenthesis that corresponds to the parenthesis pair that most closely encloses position i.

In this scenario, we may, for example, number the nodes $1, \ldots, n$ in depth-first order, and the operation `next-sibling`(i) may take a node number as an argument[2]. In this case, we need to proceed as follows:

1. Find the position of the opening parenthesis corresponding to i, say p.
2. Let $q =$ `findclose`(p).
3. Inspect the $q+1$st parenthesis of the sequence, and if it is a closing parenthesis, then return null. Otherwise, the next sibling is the node whose opening parenthesis is in at position $q + 1$.
4. Determine the number j such that the parenthesis at position $q + 1$ is the j-th opening parenthesis; return j.

Unfortunately, *all four* of the above steps require $\Omega(\log n / \log \log n)$ time if we wish to support updates to the BP string in at most poly-log n time, by reductions [6, 10] to the well-known list-ranking and subset-sum problems [11]. This appears to be a high price to pay for dynamizing, considering that operations take $O(1)$ time in the static case,

[2] This is the approach taken by the implementation of [2].

An alternative is the *finger* model [10, 19, 16], where the key object is a finger `f`, on which the following operations may be supported: (a) initialize the finger to the root of the tree (b) perform operations such as `f.op` where `op` is one of the navigation operations mentioned above. The result of an operation `f.op` is either `true`, in which case the required node (parent, next sibling etc.) exists, and the finger is moved to that node, or `false`, i.e. the node does not exist, and the finger does not move. Updates are limited to occurring in the vicinity of the finger. Interestingly, in the finger model, navigation operations can in fact be performed in $O(1)$ time [10]. While the approach proposed by [10] appears to be complex and unsuitable for a practical implementation, the simpler and practical approach of [20] that we are following appears inherently to take time $\Omega(\log n)$ for an individual operation. This raises the question: in an implementation of dynamic succinct ordinal trees based on [20], is there any practical difference between the finger model and the parenthesis-position model?

We show that the answer is "yes". For traversals, this is partly because we can largely omit steps (1), (3) and (4) above in the finger model, but also because (2) turns out to be significantly cheaper than "logarithmic" for many practical traversal seqeuences. For updates, we show that a simple strategy of "buffering" updates (only possible in the finger model) greatly improves the speed of updates for some update patterns (in our implementations we assume that we have only one finger, but the principle easily extends to multiple fingers).

Traversals vs. Individual Operations. An implementation of static succinct ordinal trees, based on the approach of [20], was reported in [2]. As with the dynamic approach of [20], the implementation of [2] also has $\Theta(\log n)$ worst-case time for individual operations. On the other hand, the implementation of [12] is in principle $O(1)$ time per operation. The starting point of our investigation was that the implementation of [2], on traversals of ordinal trees derived from some typical "benchmark" XML files, apparently had linear—rather than the expected $\Theta(n \log n)$—behaviour. The basic cause of this is that if the answer to a `findclose` or `enclose` operation is at distance d from the argument, then the answer is usually found in $O(\log d)$ rather than $O(\log n)$ time. This led us to investigate some "worst-case" trees for the implementation of [2]. While this investigation (see Section 3) produced some interesting results, the real insight was to try and directly exploit the "locality" of typical traversal sequences. As noted above, in the data structure of [20], the min-max tree built over the blocks needs to be a balanced binary tree, but the choice of balanced binary tree is not specified (however, a kind of (a, b)-tree is needed to obtain the optimal $O(\log n / \log \log n)$ time bound). The desire to exploit locality led us to consider using a *splay* tree [22], which has a number of interesting locality properties [4, 22], conjectured, others proven. In the traversals we considered, using splay trees allowed our example files to be traversed in linear observed time in some cases. The time spent on splay tree operations (measured by number of nodes accessed) appears sub-logarithmic in all the cases we considered. We do not yet have a theoretical understanding of this phenomenon.

Structure of Paper. The rest of the paper is structured as follows. Section 2 summarizes the approach of [20, 2], Section 3 summarizes our experiments on static trees, Section 4 summarizes our dynamic implementation, which is followed by an empirical evaluation in Section 5.

2 Preliminaries

Consider the BP bit string of length $2n$ where 1 represents '(' and 0 ')'. The *excess* at any position i is the number of 1's minus the number of 0's prior to position i. The excess of an opening parenthesis is also the depth of the node in the ordinal tree. We use the terms *global excess* for the excess as defined above, and *local excess* relative to some sub-string of the BP bit string for the excess of a position, measured from the start of the sub-string. We also use the term *sum* (relative to a sub-string again) for the excess at the end of the substring (which is 0 for the entire BP). We use the terms *min excess* and *max excess* to denote the minimum and maximum excess reached in a sub-string of the BP bit string. A key step in [2, 17, 20] is *excess search*, which is as follows:

- `fwd_excess`(i, rel): starts at position i going forward in the bit string searching for the leftmost node after i that has relative excess to i equal to rel;
- `bwd_excess`(i, rel): starts at position i going backward in the bit string searching for the rightmost node before i that has relative excess to i equal to rel.

Using the operations `fwd_excess`(i, rel) and `bwd_excess`(i, rel), the operations `findclose`(i), `findopen`(i) and `enclose`(i) can be implemented. For example, `findclose`$(i) =$ `fwd_excess`$(i, -1)$.

Excess Search and the Min-Max Tree. As noted above, the BP sequence is partitioned into *blocks* of size B each. These blocks are placed at the leaves of a tree such that each node contains the minimum, maximum (local) excess and sum of the concatenation of blocks under it. In the static case, this min-max tree is implemented as a binary tree stored in an array using the "heap-like" numbering; in the dynamic case, an unspecified balanced tree is recommended. An excess search starting at a block p and ending at a block $q \neq p$ will navigate up to the lowest common ancestor of p and q in the min-max tree from p, and down again to q, see [20] or Section 4 for details.

3 Traversals on Static Trees

Input trees and traversals. Although a number of real-life "benchmark" XML files are available [9], we did not use them extensively, for a variety of reasons. Firstly, the files were relatively small – the largest, although 600MB in size, had only about 25 million nodes: if stored in the information-theoretic minimum amount of space, the tree would almost completely fit into a 6MB cache.

Clearly, experiments on such trees would not give a complete account of the performance of our data structures for very large trees, particularly since cache misses are an important cause of poor performance in succinct data structures. Furthermore, our aim was to detect patterns of performance and to find "worst-case" instances. (Although not reported here, the results we got from the larger real-life benchmark XML files were in line with those we have reported.)

The experiments we performed were on four kinds of trees, of sizes approximately 64, 128, 256 and 512 million nodes. These were:

- Trees of about the above sizes obtained from XML files generated by the XMark synthetic benchmark generator (this is a standard generator for testing XML systems) [21].
- Regular k-ary trees of height h: we looked at the case $k = 2$ and $h = 25, 26, 27$ and 28 (referred to as *binary trees* henceforth), and $h = 2$ and $k = 8000, 11314, 16000$ and 22618 (henceforth *k-ary trees*).
- A "centipede" tree, where a tree with n nodes (n odd) has a path of $(n+1)/2$ nodes, with each non-root node on this path having a leaf as a right sibling.

We considered two kinds of traversals: a non-recursive depth-first traversal (DFS) and the *all root-leaf (ALR)* traversal [9], where we do DFS, but whenever the DFS encounters a leaf we trace the path back to the root.

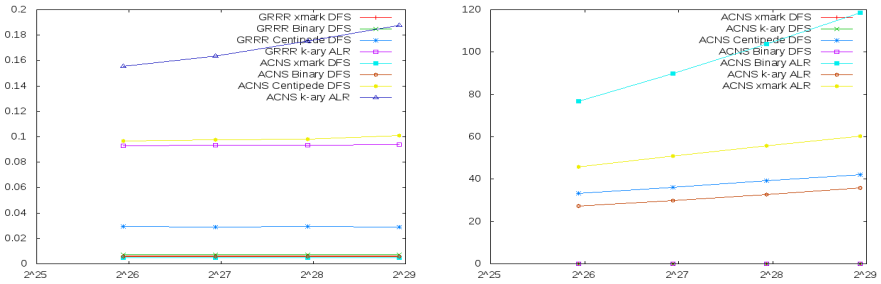

Fig. 2. Performance of the *static* implementations of Geary et al. (GRRR) and Arroyuelo et al. X-axis is the number of nodes (log-scale). (Left) Y-axis is the time for traversal divided by the number of nodes. (Right) number of tree nodes visited by Arroyuelo et al.'s implementation (ACNS) divided by the number of nodes.

Input trees and traversals. We timed the static implementations of [2, 12], code obtained from Sadakane, on the above sets of trees (see Figure 2[3]). Timings were only taken for the following pairs of inputs: XMark, Binary and Centipede with DFS, and k-ary with ALR.[4] It should be emphasized that we cannot really

[3] Some plotted data sets have y-values close to 0, hence they are not clearly visible

[4] This is mainly for succinctness, though clearly it is infeasible to run ALR on Centipede for our data sizes.

compare the two data structures against each other: the code of [2] is a 32-bit C implementation that takes about 2.3 bits/node with the default parameter settings, while the code of [12] is a 64-bit C++ implementation that takes just over twice as much space with the default parameter settings. We note that the time for DFS (per node) varies greatly with the tree for both implementations: Centipede is an order of magnitude slower. Also, for DFS on Xmark, Centipede and the binary tree, the implementation of [2] shows linear behaviour, but for ALR traversal on k-ary trees it clearly shows $\Theta(n \log n)$ behaviour. The logarithmic growth of numbers of tree nodes visited per input node is clearly visible in both the (Centipede, DFS) and (k-ary, ALR) pairs in the graph on the right. As expected in Geary et al.'s implementation, the traversal time per node is constant for all traversals.

We explain this by looking at the *traversal distance* of a traversal on a BP sequence: if the ith step of the DFS moves from a node whose parenthesis is at position p to one at position q, we set $d_i = |p - q|$, and the traversal distance is simply $\sum_i d_i$. It is easy to see that (i) the traversal distances of DFS and ARL on a k-ary tree are $\Theta(n)$ and $\Theta(n^2)$ respectively, and (ii) the traversal distance of DFS on a binary tree and Centipede is respectively $\Theta(n \log n)$ and $\Theta(n^2)$. In general it is easy to show (proof omitted):

Lemma 1. *The traversal distance of non-recursive DFS on an ordinal tree with n nodes of height h is $O(nh)$.*

Since the XMark files have a small (fixed) depth, the traversal distance for DFS on XMark files is also linear.

Using the heuristic that a navigation operation with traversal distance d usually takes $O(1+\log\lceil d/B\rceil)$ time (since the start and end points will be $\lceil d/B\rceil$ blocks apart), we see that the time spent in the min-max tree for a traversal over a tree with n nodes with overall traversal distance D would be $O(n(1 + \log\lceil D/(nB)\rceil))$. This provides some explanation of the observed data (albeit partially) – for example, we do not see a logarithmic growth in time for Arroyuelo et al.'s implementation when running DFS on Centipede. However, the traversal distance argument does suggest that in the dynamic case, a normal balanced tree (e.g. red-black) will also have relatively poor performance in cases such as ALR traversal on k-ary trees; one may get better performance by exploiting locality directly (e.g. successive leaf-to-root traversals in ALR traversals will tend to visit many min-max tree nodes in common, and have high temporal locality).

4 Engineering a Dynamic Succinct Implementation

Our *base implementation* divides the BP sequence into blocks of size B, which are leaf nodes in a binary min-max tree. Each node of the min-max tree contains the data mentioned in Section 2, see Figure 1 (right) for example. Similarly to [20, 2] we use the excess to navigate around the succinct tree and find the matches of our parentheses.

Forward excess search using the min-max tree is shown below (backward excess is similar). The function $E(i)$ returns the global excess at position i. In

Step 1 we use the length information in the min-max tree to locate the block in which parenthesis i is located, starting from the root of the tree and descending (this is needed as the number of parentheses in the blocks is not equal). The base implementation is static and we initialize by *bulk loading* it. Bulk loading splits the full BP string of a tree in equal sized blocks (except the last) and builds the min-max tree on top of them, which results in a complete balanced binary tree.

forwardExcess (i, d)

1. Use the min-max tree to locate the block in which i resides keeping track of global excess at beginning of block
2. Scan the block of i for the next parenthesis j where $i < j$ and $E(j) = E(i) + d$
   - If found return j
3. Search min-max tree for lowest common ancestor of block containing i and block containing $E(i)+d$ using minimum and maximum excess adjusting global excess at start of block while moving between nodes
4. Search min-max tree for a leaf where min excess $\leq (E(i) + d) \leq$ max excess starting from right child of lowest common ancestor, by moving to the right child of current node when $E(i) + d$ not within range of excesses of left child
5. Scan the current node and find position j such that $E(j) = E(i) + d$

Finger Model We add the finger model to our base implementation. A finger sits at a node (the *finger node*) and contains the block in which the parenthesis representing the finger node lies, its local position in the block, its current local excess and the excess at the beginning of the block. Clearly, Step 1 above is not needed in the finger model.

In addition to bulk loading, we also provide the following dynamic operations for modifying the finger model:

- `insert-first-child`(): insert a leaf as the first child of the finger node
- `insert-next-sibling`(): insert a leaf as the next sibling of the finger node

Operations `delete-first-child`() and `delete-next-sibling`() are analogous. This API can also be used to create an ordinal tree.

Implementation of Updates To insert a new leaf we shift all the parenthesis in the block to the right of the finger to make room for the leaf node. Observe that the sum of a block does not change by adding a leaf, and neither does the minimum excess. The maximum excess increases by 1, but only if a leaf is inserted at a position where the excess is already maximum. Thus, we do not need to scan the block after an insertion. Deleting a node is similar, but if the excess at the node to be deleted is the same as the current maximum excess in the block we will need to rescan the block to discover if there is another node that has the same excess, or if the block maximum excess has changed. In all cases though, the length values (if needed the excess values) of all ancestors of the block in the min-max tree have to be updated.

When the block is full, it is then split into two new blocks, each containing half of the parentheses of the previous block. To improve the space usage of

blocks (as low as 50% in the above), we implemented *incremental* copying of the blocks. We start off with a block of size B. When the block gets full we increase its capacity by a word, with an upper bound of $2B$. When the block size is $2B$ we split the block as above. Furthermore, we buffer all updates that occur in a block so long as the finger does not move to a different block. When the finger moves we *flush* the excess and length changes to all the ancestors of the block before leaving the block.

The min-max tree was implemented as a splay tree [23]. When the finger moves to a new block, then the parent of that block (which is an internal node) is splayed to the top of the tree. This results in the last accessed nodes to be closer to the root of the tree.

5 Experimental evaluation

The data structure as well as the tests were written in C++. The machine that was used to run these tests was an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz with 3MB L2 cache, running Ubuntu 10.04.1 LTS Linux and g++ 4.4.3 with optimization level 3. The Xerces-C++ 2.80 was used. Furthermore, we use the code from [2], henceforth referred to in this section as ACNS.

We first aim to justify the use of the finger model in a dynamic succinct data structures. As noted previously, the base implementation is based on the parenthesis model, and therefore each navigation operation is based upon the `findopen`(i), `findclose`(i) and `enclose`(i) operations, where i is the position of a parenthesis in the BP bit string. The first step in these operations is to start from the root of the min-max tree to locate the block in which the ith parenthesis lies. This step is unnecessary if no updates are made to a bulk-loaded tree, as in this case all blocks are equal-sized; we augment the base implementation with an array containing pointers to each block and find the block containing the ith parenthesis by indexing into this array. This is the *base + pointer array* implementation. Using the trees in 3 we perform traversals on a bulk-loaded base implementation and the base+pointer array implementation, and also on the ACNS implementation (as a "control" test).

Nodes	ACNS	Base Impl	Base + Pointer Impl
64M	0.049	0.336	0.060
128M	0.049	0.361	0.060
256M	0.049	0.383	0.060
512M	0.049	0.407	0.060

Nodes	DOM	Splay Trees
16M	46.17	31.95
32M	90.83	65.05
64M	204.04	128.37

Table 1. (Left) DFS Traversal per-node time of ACNS, "base implementation" + pointer array, "base implementation" in μs (Right) Parsing time of XMark file for DOM and Splay tree implementation. Results measure CPU time in seconds.

Table 1 presents the time it took to do a DFS traversal on XMark files of size 64, 128 and 256 million nodes. We observe from the the results of the test that ACNS and Base + Pointer implementation are showing $O(n)$ behaviour for XMark files. However, "base implementation" shows $O(n \log n)$ increase and is much slower. Since the only difference is that to access any block we need to descend the tree, it is clearly shown that this has a significant impact on the performance of the navigational operations.

For our second test, we test the speed of the insertions. Since there is no existing succinct dynamic tree implementation we compare with Xerces-C++. This is accomplished by using a SAX parser. A SAX parser will go through an XML file and raise an event when an opening/closing XML tag was encountered. We use that with our finger implementation to create an ordinal tree using `insert-first-child`, `insert-next-sibling` and we compare the parsing time by creating a Xerces-C++ DOM tree using a similar method. For the DOM case all nodes are named "a". In these tests we used XMark files up to 64M nodes. Larger sizes were not attempted due to the memory usage of DOM.

The creation time for our finger model is faster. This can be partially explained by the cache effects. Due to the small number of nodes, most of the tree will easily fit to cache in the finger model case. It was 40% faster in the case of 32 million nodes and 60% faster in the case of 64 million nodes. In that specific case the fact that the percentage was so high might have been due to the excessive memory usage of a Xerces-C++ DOM tree (for 64M nodes virtual memory exceeded 12GB). Also the speed difference seems to diverge slightly with the last test. The gap seems to be widening due to thrashing.

As a third test, we compare the traversal times for a splay tree against a bulk loaded tree. To do these we check both against 64, 128 and 256 million nodes generated by XMark, as well as with k-ary trees with similar number of nodes.

Nodes	XMark						k-ary tree					
	ACNS		Balanced		Splay		ACNS		Balanced		Splay	
	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR	DFS	ALR
64M	3.17	130.15	1.77	155.83	2.06	144.22	2.03	99.38	0.86	163.21	0.7	145.71
128M	6.35	261.25	3.54	310.76	4.15	281.45	4.05	209.60	1.74	330.36	1.41	294.29
256M	12.70	550.00	7.10	628.82	8.31	573.43	8.11	448.65	3.45	667.01	2.76	591.84

Table 2. DFS and ALR traversal comparison with using balanced tree and splay tree. Results are in seconds

From Table 2 it is clear that for DFS our data structure, both with the balanced tree and splay tree on top is faster than the ACNS data structure.This is possibly a result that was influenced from the different block sizes used between ACNS and the rest of the data structures that were traversed. Using a balanced tree appears to be faster for DFS, but when the ALR is used then Splay trees shows its possibilities, compared with the balance tree. Looking at Figure 3 we see that the splay tree will always access fewer nodes of the min-max tree, hence

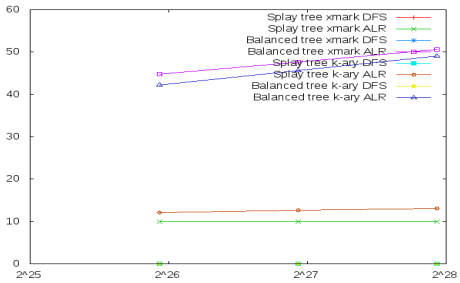

Fig. 3. Comparison of Splay trees and balanced trees. X-axis is number of nodes (log-scale). Y-axis is nodes of min-max tree visited divided by number of nodes

proving that previously traversed nodes are closer to the top so the distance to the parent that was accessed "recently" is relatively small and justifying the splay tree being faster compared to the balanced tree for ALR.

6 Conclusions

We have performed an empirical evaluation of a first implementation of dynamic succinct trees. We observe that the performance of static succinct tree implementations, particularly those based on the min-max tree which have (near-)logarithmic worst-case per-operation time complexity, is very dependent on the tree and the sequence of operations performed, more specifically, on the locality properties of the sequence of operations. By using a self-adjusting tree (the splay tree) as a basis for the min-max tree, we obtain good performance, which is arguably superior to any other balanced search tree scheme. The dynamic implementation compares well with static succinct implementations for navigation operations and with pointer-based ones for update operations. However, further work is required to expand the functionality, to better understand the effects of the splay tree on traversal performance and to investigate implementations of $O(1)$-time dynamic succinct trees.

References

1. Apache: Xerces-c++ xml parser (Jan 2012), http://xerces.apache.org/xerces-c/
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Blelloch, G.E., Halperin, D. (eds.) ALENEX. pp. 84–97. SIAM (2010)
3. Arroyuelo, D., Claude, F., Maneth, S., Mäkinen, V., Navarro, G., Nguyen, K., Sirén, J., Välimäki, N.: Fast in-memory xpath search using compressed indexes. In: ICDE. pp. 417–428 (2010)
4. Badoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theor. Comput. Sci. 382(2), 86–96 (2007)
5. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)

6. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
7. Delpratt, O.: The sixml project (Mar 2010), http://www.cs.le.ac.uk/SiXML/
8. Delpratt, O., Rahman, N., Raman, R.: Engineering the louds succinct tree representation. In: Àlvarez, C., Serna, M.J. (eds.) WEA. Lecture Notes in Computer Science, vol. 4007, pp. 134–145. Springer (2006)
9. Delpratt, O., Raman, R., Rahman, N.: Engineering succinct dom. In: EDBT. pp. 49–60 (2008)
10. Farzan, A., Munro, J.I.: Succinct representation of dynamic trees. Theor. Comput. Sci. 412(24), 2668–2678 (2011), prelim. version *ICALP '09*.
11. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Johnson, D.S. (ed.) STOC. pp. 345–354. ACM (1989)
12. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368(3), 231–246 (2006)
13. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
14. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. pp. 549–554. IEEE Computer Society (1989)
15. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
16. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: SODA. pp. 529–536 (2001)
17. Munro, J.I., Rao, S.S.: Succinct representations of functions. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP. Lecture Notes in Computer Science, vol. 3142, pp. 1006–1015. Springer (2004)
18. Munro, J.I., Rao, S.S.: Handbook of Data Structures and Applications, chap. 37, Succinct Representation of Data Structures. Chapman & Hall/CRC (2004)
19. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP. Lecture Notes in Computer Science, vol. 2719, pp. 357–368. Springer (2003)
20. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Charikar, M. (ed.) SODA. pp. 134–149. SIAM (2010)
21. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data management. In: VLDB. pp. 974–985. Morgan Kaufmann (2002)
22. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
23. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM (1987)
24. W3C: Document object model (Jan 2009), http://www.w3.org/DOM/
25. Wong, R.K., Lam, F., Shui, W.M.: Querying and maintaining a compact xml storage. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) WWW. pp. 1073–1082. ACM (2007)