A Practical Succinct Data Structure for Tree-Like Graphs

Johannes Fischer[1] and Daniel Peters[2]

[1] TU Dortmund, Germany
johannes.fischer@cs.tu-dortmund.de
[2] Physikalisch-Technische Bundesanstalt (PTB), Germany
daniel.peters@ptb.de

Abstract. We present a new succinct data structure for graphs that are "tree-like," in the sense that the number of "additional" edges (w.r.t. a spanning tree) is not too high. Our algorithmic idea is to represent a BFS-spanning tree of the graph with a succinct data structure for trees, and enhance it with additional information that accounts for the non-tree edges. In practical tests, our data structure performs well for graphs containing up to 10% of non-tree edges, reducing the space of a pointer-based representation by a factor of ≈20, while increasing the worst-case running times for the operations by roughly the same factor.

1 Introduction

Succinct data structures have been one of the key contributions to the algorithms community in the past two decades. Their goal is to represent objects from a universe of size u in information-theoretical optimal $\lg u$ bits of space.[1] Apart from the bare representation of the object, fast operations should also be supported, ideally in time no worse than with a "conventional" data structure for the object. For this, one usually allows extra space $o(\lg u)$ bits.

A prime example of succinct data structures are ordered rooted trees, where with n nodes we have $u \approx 4^n$. In 1989, Jacobson made a first step towards achieving this goal, by giving a data structure using $10n + o(n)$ bits, while supporting the most common navigational operations in $O(\lg n)$ time [19]. This was further improved to the optimal $2n + o(n)$ bits and optimal $O(1)$ navigation time by Munro and Raman [25]. Note that a conventional, pointer-based data structure for trees requires $\Theta(n \lg n)$ bits, which is off by a factor of $\lg n$ from the information-theoretical minimum.

Since the work of Munro and Raman, the research on succinct data structures has blossomed. We now have succinct data structures for bit-vectors [27], permutations [23], binary relations [2], dictionaries [26], suffix trees [29], to name just a few.

The practical value of those data structures has sometimes been disputed. However, as far as we know, in all cases where genuine attempts were made at

[1] Function lg denotes the binary logarithm throughout this paper.

M.S. Rahman and E. Tomita (Eds.): WALCOM 2015, LNCS 8973, pp. 65–76, 2015.

practical implementations, the results have mostly been successful [13,20,16, etc., to cite some recent papers presented in the algorithm engineering community]. Further examples of well-performing practical succinct tree implementations will be mentioned throughout this paper.

1.1 Our Contribution

We focus on the succinct representation of a very practical class of graphs: graphs that are "tree-like" in the sense that the number of edges, which can potentially be $\Theta(n^2)$ for an n-node graph, is much lower. We measure this tree-likeness by introducing two additional parameters: (1) k, the number of "additional" edges that have to be added to a spanning tree of the graph (note that $k = m - n + 1$ if m denotes the total number of edges), and (2) h, the number of nodes having more than one incoming edge (also called non-tree nodes in the following). This definition of tree-likeness is similar in flavor to the k-almost trees by Gurevich et al. [17], but in the latter the number of additional edges is counted separately for each biconnected component, with k being the maximum of these.

We think that our definition of tree-likeness encompasses a large range of instances arising in practice. One important example comes from computational biology, where one models the ancestral relationships between species by phylogenetic trees. However, sometimes there are also non-bifurcating specification events [18]. One approach to handle those events are phylogenetic networks, which have an underlying tree as a basis, but with added cross-edges to model the passing of genetic material that does not follow the tree.

Our first contribution (Sect. 3) is a theoretical formulation of a succinct data structure for graphs with the above mentioned parameters n, m, k, and h. It uses space at most $(2n + m) \lg 3 + h \lg n + k \lg h + o(m + k \lg h) + O(\lg \lg n)$ bits, which is close to the $2n + o(n)$ bits for succinct trees if k (and hence also m and h) is close to n. This should be compared to the $O((n + m) \lg n)$ bits that were needed if the graph was represented using a pointer-based data structure. Our second contribution is that we show that the data structure is amenable to a practical implementation (Sect. 4–5). We show that we can reduce the space from a conventional pointer-based representation by a factor of about 20, while the times for navigational operations (moving in either direction of the edges) increase by roughly the same factor; such a space-time tradeoff is typical for succinct data structures.

1.2 Further Theoretical Work on Succinct Graphs

Farzan and Munro [9] showed how to represent a general graph succinctly in $\lg \binom{n^2}{m}(1+o(1))$ bits of space, while supporting the operations supported both by adjacency lists and by adjacency matrices in optimal time. Other results exist for special types of graphs: separable graphs [5], planar graphs [25], pagenumber-k graphs [11], graphs of limited arboricity [21], and DAGs [8]. However, to the best of our knowledge, only the approach on separable graphs has been implemented

so far [6]. Also, none of the approaches can navigate efficiently to the sources of the *incoming* edges (without doubling the space), as we do.

2 Preliminaries

In this section we introduce existing data structures that form the basis of our new succinct graph representation. All these results (hence also our new one) are in the word-RAM model of computation, where it is assumed that the machine consists of words of width w bits that can be manipulated in $O(1)$ time by a standard set of arithmetic and logical operations, and further that the problem size n is not larger than $O(2^w)$.

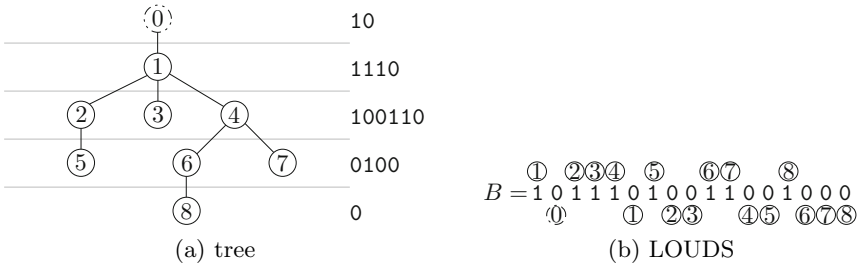
2.1 Succinct Data Structures

Let $S[0, n)$ be a *bit-string* of length n. We define the fundamental *rank*- and *select*-operations on S as follows: $\mathsf{rank}_1(S, i)$ gives the number of `1`'s in the prefix $S[0, i]$, and $\mathsf{select}_1(S, i)$ gives the position of the i'th `1` in S, reading S from left to right ($0 \leq i < n$). Operations $\mathsf{rank}_0(S, i)$ and $\mathsf{select}_0(S, i)$ are defined similarly for `0`-bits. S can be represented in $n + o(n)$ bits such that rank- and select-operations are supported in $O(1)$ time [25].

These operations have been extended to sequences over larger alphabets, at the cost of slight slowdowns in the running times [14]: let $S[0, n)$ be a *string* over an alphabet Σ of size σ. Then S can be represented in $n \lg \sigma(1 + o(1))$ bits of space such that the operations $\mathsf{rank}_a(S, i)$ and $S[i]$ (accessing the i'th element) take $O(\lg \lg \sigma)$ time, and $\mathsf{select}_a(S, i)$ takes $O(1)$ time (all for arbitrary $a \in \Sigma$ and arbitrary $0 \leq i < n$). Note that by additionally storing S in plain form, the access-operation also takes $O(1)$ time, at the cost of duplicating the space. In some special cases the running times for the three operations is faster. For example, when the alphabet size is small enough such that $\sigma = w^{O(1)}$ for word size w, then Belazzougui and Navarro [3] proved that $O(1)$ time for all three operations is possible within $O(n \lg \sigma)$ bits of space.

2.2 The Level Order Unary Degree Sequence (LOUDS)

There are several ways to represent an ordered tree on n nodes using $2n$ bits [24, 4]; in this article, we focus on one of the oldest approaches, the *level order unary degree sequence* [19], which is obtained as follows (the reasons for preferring LOUDS over BPS [24] or DFUDS [4] will become evident when introducing the new data structure in Sect. 3). For convenience, we first augment the tree with an artificial *super-root* that is connected with the original root of the tree. Now initialize B as an empty bit-vector and traverse the nodes of the tree level by level (aka breadth-first). Whenever we see a node with k children during this level-order traversal, we append the bits $\mathtt{1}^k\mathtt{0}$ to S, where $\mathtt{1}^k$ denotes the juxtaposition of k `1`-bits. See Fig. 1 for an example. In the LOUDS, each node is represented twice: once by a '`1`,' written when the node was seen as a *child*



(a) tree (b) LOUDS

Fig. 1. An ordered tree (a) and its level order unary degree sequence (b)

during the level-order traversal, and once by a '0,' written when it was seen as a *parent*. The number of bits in B is $2n + 1$.

We identify the nodes with their level-order number, since both the 1- and the 0-bits appear in this order in B. It should be noted that all succinct data structures for trees [19, 24, 4, 10, 7] must have the freedom to fix a particular naming for the nodes; natural such namings are post- or pre-order [19, 24, 4], in-order [7], and level-order [19], as here.[2]

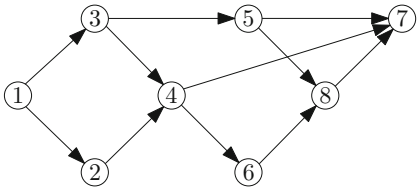
If we now augment B with data structures for rank and select (see Sect. 2.1), then the resulting space is $2n + o(n)$ bits, but basic navigational operations on the tree can be *simulated* in $O(1)$ time: for moving to the parent node of i ($1 \leq i \leq n$), we jump to the position y of the i'th 1-bit in B by $y = \mathsf{select}_1(B, i)$, and then count the number j of 0's that appear before y in B by $j = \mathsf{rank}_0(B, y)$; j is then the level-order number of the parent of i. Conversely, listing the children of i works by jumping to the position x of the i'th 0-bit in B by $x = \mathsf{select}_0(B, i)$, and then iterating over the positions $x+1, x+2, \ldots$, as long as the corresponding bit is '1.' For each such position $x+k$ with $B[x+k] = 1$, the level-order numbers of i's children are $\mathsf{rank}_1(B, x) + k$, which can be simplified to $x - i + k + 1$.

3 New Data Structure

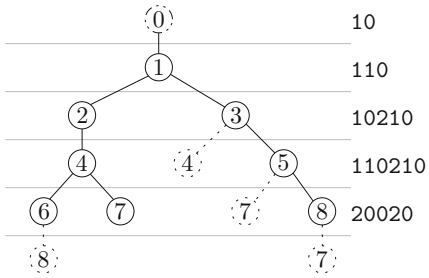
We now propose our new succinct data structure for tree-like graphs. Let G denote a directed graph. We use the following characteristics of G:

- n, the number of nodes in G,
- m, the number of edges in G,
- $c \leq n$, the number of strongly connected components with no incoming edge from a different strongly connected component,
- $k = m - n + 1$, the number of non-tree edges in G (the number of edges to be added to a spanning tree of G to obtain G), and
- $h \leq k$, the number of non-tree nodes in G (nodes with more than 1 incoming edge).

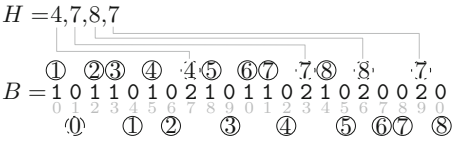
[2] If the naming is arbitrary (e.g., chosen by the user), then $n \lg n$ bits are inevitable, since *any* memory layout of the nodes has $n!$ possible namings.



(a) A graph G



(b) The resulting tree T_G. Shadow nodes are depicted as dashed lines.



(c) The resulting ternary LOUDS B and the array H of non-tree edges

Fig. 2. Illustration of our new data structure. The nodes are numbered such that they correspond to the level-order numbers in the chosen BFT-tree.

For simplicity, assume for now that there exists a node r in G from which a path to every other node exists (i.e., $c = 1$). From r, perform a breadth-first traversal (BFT) of G. Let T_G^{BFT} denote the resulting BFT-tree. We augment T_G^{BFT} as follows: for each node w that is *inspected but not visited* during the BFT at node v (meaning that it has already been visited at an earlier point), we make a *copy* of w and append it as a child of v in the BFT-tree T_G^{BFT}. We call those nodes *shadow nodes*. Finally, we add a super-root to r, and call the resulting tree T_G, which has exactly $m+2$ nodes. See Fig. 2a and 2b for an example of G and T_G. If no such node r exists, we perform the BFT from c nodes $r_1, \ldots, r_c$ for every strongly connected component with no incoming edge from a different such component, and obtain a BFT-*forest*. All roots of this forest will be made children of the super-root. This adds at most c additional edges to T_G.

We now aim at representing the tree T_G space efficiently, similar to the LOUDS of Sect. 2.2. Since we need to distinguish between real nodes and shadow nodes, we cannot construct a *bit*-vector anymore. Instead, we construct B as a sequence of *trits*, namely values from $\{0, 1, 2\}$, as follows: again, B is initially empty, and we visit the nodes of T_G in level-order. For each visited node, the sequence appended to B is constructed as in the original LOUDS, but now using a '2' instead of a '1' for shadow nodes. The shadow nodes are *not* visited again during the level-order traversal and hence *not* represented by 0's.[3] We call the resulting

[3] Listing the shadow nodes by 0's would not harm, but does not yield any extra information; hence we can omit them.

Function `children(`i`)`: find the nodes directly reachable from i.

```
x ← select_0(B, i) + 1;                    // start of the list of i's children
while B[x] ≠ 0 do
    if B[x] = 1 then output rank_1(B, x);                    // actual node
    else output H[rank_2(B, x)];                             // shadow node
    x ← x + 1;
endw
```

Function `parents(`i`)`: find the nodes from which i is directly reachable.

```
output rank_0(B, select_1(B, i));                            // tree parent
j ← 1;
x ← select_i(H, j);
while x < k do
    output rank_0(B, select_2(B, x + 1));                    // non-tree parent
    j ← j + 1;
    x ← select_i(H, j);
endw
```

trit-vector B the *ternary LOUDS*. The ternary LOUDS consists of $n+m+c+1$ trits. See Fig. 2c for an example.

We also need an additional array H that lists the non-tree nodes in the order in which they appear in B. This array will be used for the navigational operations, as shown in Sect. 3.1. For the operations, besides accessing H, we will also need select-support on H. For this, we use the data structures mentioned in Sect. 2.1 [3, 14].

3.1 Algorithms

The algorithms for listing the children and parents of a node are shown in Functions children(i) and parents(i). These functions follow the original LOUDS-functions as closely as possible. Listing the children just needs to make the distinction if there is a '1' or a '2' in the ternary LOUDS B; in the latter case, array H storing the shadow nodes needs to be accessed.

Listing the parents is only slightly more involved. First, the (only) tree parent can be obtained as in the original LOUDS. Then we iterate through the occurrences of i in H in a while-loop, using select-queries. For each occurrence found, we go to the corresponding '2' in B and count the number of '0's before that '2' as usual.

As in the original LOUDS, counting the number of children is faster than traversing them: simply calculate $\mathsf{select}_0(B, i+1) - \mathsf{select}_0(B, i) + 1$; this computes the desired result in $O(1)$ time.[4]

[4] For calculating the number of parents in $O(1)$ time, we would need to store those numbers explicitly for hybrid nodes; for all other nodes it is 1.

3.2 Space Analysis

The trit-vector B can be stored in $(n + m + c)(\lg 3 + o(1))$ bits [27], while supporting $O(1)$ access on its elements. Support for rank and select-queries needs additional $o(n + m)$ bits [25].

There are several ways to store H. Storing it in plain form uses $k \lg n$ bits. Using another $k \lg n(1 + o(1))$ bits, we can also support $\mathsf{select}_a(H, i)$-queries on H in constant time [14]. This sums up to $2k \lg n + o(k \lg n)$ bits.

On the other hand, since the number h of non-tree nodes can be much smaller than k (the number of non-tree edges), this can be improved with a little bit of more work: we store a translation table $T[0, h)$ such that $T[i]$ is the level order number of the i'th non-tree node. Then $H[0, k)$ can be implemented by a table $H'[0, k)$ that only stores values from $[0, h)$, such that $H[i] = T[H'[i]]$. The space for T and H' is $k \lg h + h \lg n$ bits. To also support select-queries on H within less than $k \lg n$ bits of space, we use the *indexable dictionaries* of Raman et al. [28]: store a bit vector $C[0, n)$ such that $C[i] = \mathtt{1}$ iff the i'th node in level order is a non-tree node. C can be stored in $h \lg n + o(h) + O(\lg \lg n)$ bits [28, Thm. 3.1], while supporting select- and partial rank-queries (only $\mathsf{rank}_\mathtt{1}(C, i)$ with $C[i] = \mathtt{1}$, which is what we need here) in constant time. Now we only need to prepare H' for select-queries, this time using $k \lg h + o(k \lg h)$ bits. Queries $\mathsf{select}_a(H, i)$ can be answered by $\mathsf{select}_{\mathsf{rank}_\mathtt{1}(C,a)}(H', i)$, so H can be discarded. Since the data structure of Raman et al. [28] automatically supports select-queries, we also do not need to store T in plain form anymore, since $T[i] = \mathsf{select}_\mathtt{1}(C, i)$. Thus, the total space for H using this second approach is $h \lg n + k \lg h + o(h + k \lg h) + O(\lg \lg n)$ bits.

Summing up and simplifying ($c \leq n$), the main theoretical result of this article can be formulated as follows:

Theorem 1. *A directed graph G with n nodes, m edges, and h non-tree nodes ($k = m - n + 1$ is the number of non-tree edges) can be represented in*

$$(2n + m) \lg 3 + h \lg n + k \lg h + o(m + k \lg h) + O(\lg \lg n)$$

bits such that listing the x incoming or y outgoing edges of any node can be done in $O(x)$ or $O(y)$ time, respectively. Counting the number of outgoing edges can be done in $O(1)$ time.

4 Implementation Details

We now give some details of our implementation of the data structure from Sect. 3, sometimes sacrificing theoretical worst-case guarantees for better results in practice.

4.1 Representing Trit-Vectors

We first explain how we store the trit sequence B such that constant time access, rank and select are supported. We group 5 trits together into one tryte, and store

this tryte in a single byte. This results in space $\lceil (n+m+c)/5 \rceil \cdot 8 = \lceil 1.6\,(n+m+c) \rceil$ bits for B, which is only $\approx 1\%$ more than the optimal $\lceil (n+m+c) \lg 3 \rceil \approx \lceil 1.585\,(n+m+c) \rceil$ bits. The individual trits are reconstructed using Horner's method, in just one calculation.[5]

For rank and select on B, we use an approach similar to the *bit*-vectors of González et al. [15], but with a three-level scheme (instead of only 2), thus favoring space over time. This scheme basically stores rank-samples at increasing sample rates, and the fact that the bits are now intermingled with `2`'s does not cause any troubles. We used sample rates 25, 275, and 65 725 trits, respectively, which enable a fast byte-aligned layout in memory. On the smallest level we divided a 25-trit block into five trytes. Using the table lookup technique [22] on the trytes the calculation for rank on a 25-trit block is done in at most five steps with an overhead of $3^5 = 243$ bytes of space.

As in the original publication [15], select queries are solved by binary searches on rank-samples, again favoring space over time.

4.2 Other Data Structures

Instead of the complex representation of H as described in Sect. 3.2, needed for an efficient support of the parent-operation, we used a simpler array-based approach: we store the positions of `1`-bits (in B) of the first occurrences of non-tree nodes in an array $P[0,h)$. (In the example of Fig. 2, we have $P = [5, 11, 14]$ for the non-tree nodes 4,7, and 8.) A second array $Q[0,k)$ lists the positions of the other occurrences of the non-tree nodes, in level order (In the example, $Q = [7; 13, 19; 16]$). A final third array $N[0,h)$ stores the starting positions of the non-tree nodes in Q (in the example, $N = [0, 1, 3]$). Then with a binary search on P (or a bit-vector marking the respective positions) we can find out if a node i has further shadow copies, and if so, list them using Q and N. Note that with these arrays, we can also efficiently list (in $O(1)$ time) the number of parents of non-tree nodes.

We also added a bit-vector $D = [0,n)$ with $D[i] = 1$ iff node i is a leaf node. This way, the question if a node has children can be quickly answered by just one look-up to D, omitting rank and select queries.

5 Practical Results

The aim of this section is to show the practicality of our approach on the example of phylogenetic networks. Such networks arise in computational biology. They are a generalization of the better known phylogenetic trees, which model the (hypothetic) ancestral relationships between species. In particular for fast reproducing organisms like bacteria, networks can better explain the observed data

[5] We did not investigate codes that exploit the fact that the distribution of the `0`'s, `1`'s, and `2`'s in B is not necessarily uniform. Some further space could be saved here, probably at the cost of increased access times. We leave this as a direction for future research.

than trees. Quoting Huson and Scornavacca [18], phylogenetic networks "may be more suitable for data sets where evolution involves significant amounts of reticulate events, such as hybridization, horizontal gene transfer, or recombination."

Since large real-life networks are not (yet) available, we chose to create them artificially for our tests. We did so by creating random tree-like graphs with 10% non-tree edges ($k = n/10$), by *directly* creating random trit-vectors of a given length, and randomly introducing k 2's to create non-tree edges. We further ensured that shadow nodes have different parents, and that all non-tree edges point only to nodes at the same height (in the BFS-tree), mirroring the structure of phylogentic networks (no interchange of genetic material with extinct species).

We compared our data structure to a conventional pointer-based data structure for graphs (where each node stores a list of its descendants, a pointer to an arbitrary father, and the number of its descendants). While there exist many implementations of succinct data structures for trees[6], we are not aware of any implementations for graphs, hence we did not compare our data structure to others.

Our machine was equipped with an Intel Core i7@2.2GHz and 8GB of RAM, running under Windows 7. We compiled the program for 32 bits, in order not to make the pointer-based representation unnecessarily large. All programs used only a single core of the CPU. We averaged the running times over 1 000 tests for $n = 10\,000$, over 100 tests for $100\,000 \leq n \leq 1\,000\,000$, over 15 tests for $n = 10\,000\,000$, and over 5 tests for $n = 100\,000\,000$.

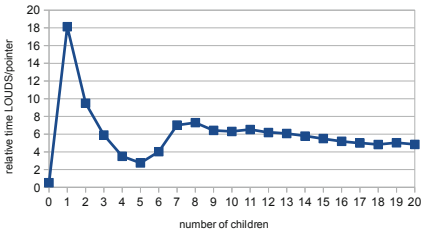
Table 1. Comparison between a pointer based graph and our succinct LOUDS representation for graphs with 10% non-tree edges

	space [MByte]		time for children [μsec]		time for parents [μsec]	
n	LOUDS	pointer	LOUDS	pointer	LOUDS	pointer
10 000	0.0159	0.3654	0.3203	0.0295	0.3315	0.0129
100 000	0.1682	3.6533	0.3458	0.0311	0.3472	0.0130
1 000 000	1.6818	36.5433	0.3884	0.0332	0.3614	0.0136
10 000 000	18.8141	365.4453	0.3889	0.03374	0.3812	0.0138
100 000 000	188.1542	3 654.4394	0.4095	—	0.4198	—

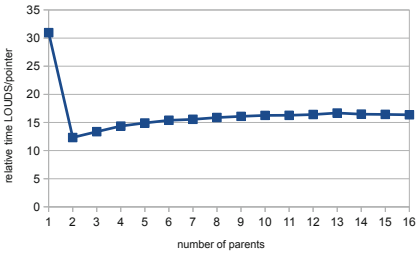
Table 1 shows the sizes of the data structures and the average running times for the children- and parents-operations with either representation.[7] It can be seen that our data structure is consistently about 20–25 times smaller than the pointer-based structure, while the time for the operations increases by a factor

[6] For example, the well-known libraries for succinct data structures `https://github.com/fclaude/libcds` and `https://github.com/simongog/sdsl` both have well-tuned succinct tree implementations. Other sources are [1, 12].

[7] For memory reasons, the running times of the pointer-based representation could not be measured for the last instances.



(a) Listing the children of a node. The graph shows the relative slow-down of our LOUDS over a pointer-based representation for nodes with varying number of children.



(b) The same as in (a), but now for listing the parents of a node

Fig. 3. Detailed evaluation of running times

of about 12 in case of the children-operation, and by a factor of about 25 in case of the parents-operation. Such trade-offs are typical in the world of succinct data structures.

To further evaluate our data structure, we more closely surveyed the children- and parents-operations in a graph with 1 000 000 nodes and 10% of non-tree edges, in which a node has no more than 16 incoming edges. We executed both operations on every node in the graph and grouped the running times by the number of children and parents, respectively. The results are shown in Fig. 3. In (a), showing the results for the children-operations, several interesting points can be observed. First, for nodes with 0 children (a.k.a. leaves), our data structure is actually *faster* than the pointer-based representation (about twice as fast), because this operation can be answered by simply checking one bit in the bit-vector D, mentioned in Sect. 4. Second, for nodes with 5 children the slowdown is only about 3, then rises to a slowdown of about 7 for nodes with 8 children, and finally gradually levels off and seems to convert to a slowdown of about 5. We think that this can be explained by the different distributions of the *types* of the nodes listed in the children operation: while for tree-nodes the node numbers can be simply calculated from the LOUDS, for non-tree nodes this process involves further look-ups, e.g. to the H-array. Since we tested graphs with 10% non-tree edges, we think that at about 7–8 children/nodes this effect is most expressed. In (b) the parents operation on our LOUDS for nodes with one parent is around 30 times slower than the pointer representation. For a greater number of parents it is about 16 times slower. Our explanation is that at first a rank and select query is necessary to retrieve the first parent node, afterwards if the node has more than one parent the H-array is scanned. With our practical implementation of the H-array from Sect. 4 the select results are directly saved in the Q-array, hence there is no need for select queries anymore and a rank query seems to be around 16 times slower than a look-up.

6 Conclusions

We presented a framework and implementation for a new succinct data structure for "tree-like" graphs based on the LOUDS representation for trees. The practical evaluation confirmed that our succinct data structure achieves a significant space reduction. A trade-off between space and time can be observed, which is common in the world of succinct data structures.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. ALENEX, pp. 84–97. SIAM (2010)
2. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 170–183. Springer, Heidelberg (2010)
3. Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 181–192. Springer, Heidelberg (2012)
4. 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)
5. Blandford, D.K., Blelloch, G.E., Kash, I.A.: Compact representations of separable graphs. In: Proc. SODA, pp. 679–688. ACM/SIAM (2003)
6. Blandford, D.K., Blelloch, G.E., Kash, I.A.: An experimental analysis of a compact graph representation. In: ALENEX/ANALC, pp. 49–61. SIAM (2004)
7. Davoodi, P., Raman, R., Satti, S.R.: Succinct representations of binary trees for range minimum queries. In: Gudmundsson, J., Mestre, J., Viglas, T. (eds.) COCOON 2012. LNCS, vol. 7434, pp. 396–407. Springer, Heidelberg (2012)
8. Farzan, A., Fischer, J.: Compact representation of posets. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 302–311. Springer, Heidelberg (2011)
9. Farzan, A., Munro, J.I.: Succinct representations of arbitrary graphs. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 393–404. Springer, Heidelberg (2008)
10. Farzan, A., Munro, J.I.: A uniform approach towards succinct representation of trees. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 173–184. Springer, Heidelberg (2008)
11. Gavoille, C., Hanusse, N.: On compact encoding of pagenumber k graphs. Discrete Mathematics & Theoretical Computer Science 10(3), 23–34 (2008)
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. Gog, S., Ohlebusch, E.: Fast and lightweight LCP-array construction algorithms. In: Proc. ALENEX, pp. 25–34. SIAM (2011)
14. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. SODA, pp. 368–373. ACM/SIAM (2006)
15. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA), Greece, pp. 27–38. CTI Press and Ellinika Grammata (2005)

16. Grossi, R., Ottaviano, G.: Design of practical succinct data structures for large data collections. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 5–17. Springer, Heidelberg (2013)
17. Gurevich, Y., Stockmeyer, L., Vishkin, U.: Solving NP-hard problems on graphs that are almost trees and an application to facility location problems. J. ACM 31(3), 459–473 (1984)
18. Huson, D.H., Scornavacca, C.: A survey of combinatorial methods for phylogenetic networks. Genome Biology and Evolution 3, 23 (2011)
19. Jacobson, G.J.: Space-efficient static trees and graphs. In: Proc. FOCS, pp. 549–554. IEEE Computer Society (1989)
20. Joannou, S., Raman, R.: Dynamizing succinct tree representations. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 224–235. Springer, Heidelberg (2012)
21. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. SIAM J. Discrete Math. 5(4), 596–603 (1992)
22. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
23. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 345–356. Springer, Heidelberg (2003)
24. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: Proc. FOCS, pp. 118–126. IEEE Computer Society (1997)
25. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
26. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Comput. 31(2), 353–363 (2001)
27. Pătraşcu, M.: Succincter. In: Proc. FOCS, pp. 305–313. IEEE Computer Society (2008)
28. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. ACM Transactions on Algorithms 3(4), Article No. 43 (2007)
29. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst 41(4), 589–607 (2007)