Journal of Discrete Algorithms 36 (2016) 39–49

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

GLOUDS: Representing tree-like graphs

Johannes Fischer [a,*], Daniel Peters [b]

[a] Department of Computer Science, TU Dortmund, Germany
[b] Physikalisch-Technische Bundesanstalt (PTB), Germany

ARTICLE INFO

Article history:
Available online 4 November 2015

Keywords:
Succinct data structures

ABSTRACT

The Graph Level Order Unary Degree Sequence (GLOUDS) is a new succinct data structure for directed graphs that are "tree-like," in the sense that the number of "additional" edges (w.r.t. a spanning tree) is not too high. The algorithmic idea is to represent a BFS-spanning tree of the graph (consisting of n nodes) with a well known succinct data structure for trees, named LOUDS, 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 $m = 5n$ edges, while still having competitive running times for listing adjacent nodes.

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 space $\lg u$ 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 [25].

A prime example of succinct data structures are ordered rooted trees, where with n nodes we have $\lg u \approx 2n$. 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 [25]. This was further improved to the asymptotically optimal $2n + o(n)$ bits and optimal $O(1)$ navigation time by Munro and Raman [35]. 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 [37], permutations [33], binary relations [2], dictionaries [36], suffix trees [39], 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 practical implementations, the results have mostly been successful [18,26,22, 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.

* Corresponding author.
E-mail addresses: johannes.fischer@cs.tu-dortmund.de (J. Fischer), daniel.peters@ptb.de (D. Peters).

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

http://dx.doi.org/10.1016/j.jda.2015.10.004

1.1. Our contribution

We focus on the succinct representation of a very practical class of directed 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. $h \leq k$, the number of nodes having more than one incoming edge (also called non-tree nodes in the following).

Our focus are graphs where $k = O(n)$, with a small constant in the big-O. This definition of tree-likeness is similar in flavor to the k-almost trees by Gurevich et al. [23], 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 [24]. 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. Another example of tree-like graphs are *compact directed acyclic word graphs* (CDAWGS), a well-known text indexing data structure [7].

Our first contribution (Section 3) is a theoretical formulation of the GLOUDS, 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 (Sections 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 [14] 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 [35], pagenumber-k graphs [16], graphs of limited arboricity [27], and DAGs [13]. 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.

For a good overview of the theoretical work on succinct graph representation, see the recent survey by Munro and Nicholson [32].

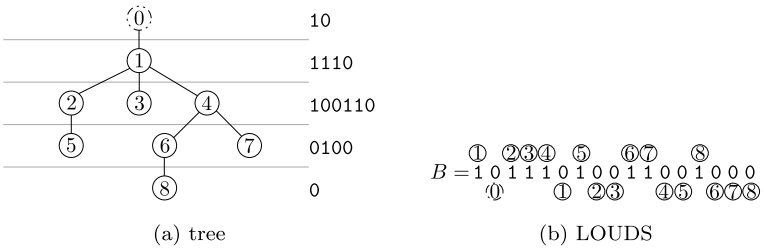
2. Preliminaries

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 $S[0, i]$ $(0 \leq i < n)$, and $\mathsf{select}_1(S, i)$ gives the position of the i'th `1` in S, reading S from left to right $(1 \leq i \leq n)$. 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 [25,31].

These operations have been extended to sequences over larger alphabets, at the cost of slight slowdowns in the running times [19]: 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 doubling 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.



(a) tree (b) LOUDS

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

2.2. The level order unary degree sequence (LOUDS)

There are several ways to represent an ordered tree on n nodes using $2n$ bits [34,4]; in this article, we focus on one of the oldest approaches, the *level order unary degree sequence* [25], which is obtained as follows (the reasons for preferring LOUDS over BPS [34] or DFUDS [4] will become evident when introducing the new data structure in Section 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 B, where $\mathtt{1}^k$ denotes the juxtaposition of k $\mathtt{1}$-bits. See Fig. 1 for an example. In the resulting LOUDS, each node is represented twice: once by a '1,' written when the node was seen as a *child* 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 $\mathtt{1}$- and the $\mathtt{0}$-bits appear in this order in B. It should be noted that all succinct data structures for trees [25,34,4,15,12] must have the freedom to fix a particular naming for the nodes; natural such namings are post- or pre-order [25,34,4], in-order [12], and level-order [25], as here.[2]

If we now augment B with data structures for rank and select (see Section 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 ($1 \le i \le n$), we jump to the position y of the i'th $\mathtt{1}$-bit in B by $y = \mathsf{select}_1(B, i)$, and then count the number j of $\mathtt{0}$'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 $\mathtt{0}$-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 $B[x+k] = \mathtt{1}$, 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. GLOUDS

We now propose our new succinct data structure for tree-like graphs, which we called *graph level order unary degree sequence* (GLOUDS).

Let G denote a directed graph. We use the following characteristics of G:

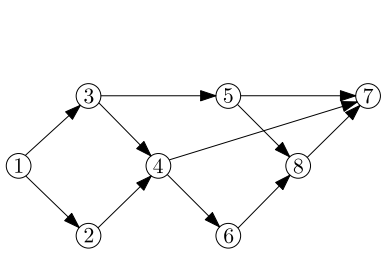
- n, the number of nodes in G,
- m, the number of edges in G,
- $c \le n$, the number of *roots* in G, i.e., the size of a minimal set of nodes from which directed paths to all other nodes exist [28, p. 372],
- $k = m - n + 1$, the number of non-tree edges in G (the number of edges to be added to a spanning tree of G to obtain G), and
- $h \le k$, the number of non-tree nodes in G (nodes with more than 1 incoming edge).

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.

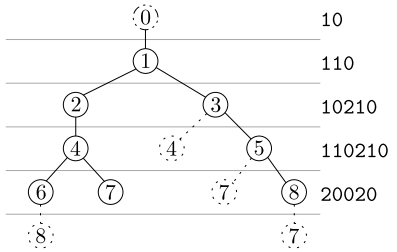
If no such node r exists, we perform the BFT from c roots $r_1, \ldots, r_c$, and obtain a BFT-*forest*. All roots of this forest will be made children of the super-root. This adds at most c additional edges to T_G.

We now aim at representing the tree T_G space efficiently, similar to the LOUDS of Section 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 $\{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$, as follows: again, B is initially empty, and we visit the nodes of T_G in

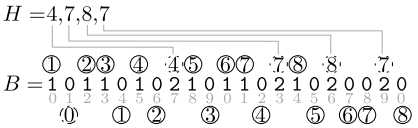
[2] If the naming is arbitrary (e.g., chosen by the user), then $n \lg n$ bits are inevitable, since *any* memory layout of the nodes has $n!$ possible namings.



(a) A graph G.



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



(c) The resulting GLOUDS 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.

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) − 1];             // shadow node
    x ← x + 1;
end while
```

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

We also need an additional array $H[0,k)$ 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 Section 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 Section 2.1 [3,19].

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 GLOUDS 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]

3.2. Space analysis

The trit-vector B can be stored in $(n+m+c)(\lg 3 + o(1))$ bits [37], while supporting $O(1)$ access on its elements. Support for rank and select-queries needs additional $o(n+m)$ bits [25,31].

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 [19]. 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

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

[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.

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);
end while
```

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 combined 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 can use the *indexable dictionaries* of Raman et al. [38]: store a bit vector $C[0,n)$ such that $C[i] = 1$ 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 [38, Thm. 3.1], while supporting select- and partial rank-queries (only $\mathsf{rank}_1(C,i)$ with $C[i] = 1$, 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 $\mathsf{select}_{\mathsf{rank}_1(C,a)}(H',i)$, so H can be discarded. Since the data structure of Raman et al. [38] automatically supports select-queries, we also do not need to store T in plain form anymore, since $T[i] = \mathsf{select}_1(C,i)$. 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 ($c \le n$), 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 Section 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. [20], 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 [31] 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 [20], select queries are solved by binary searches on rank-samples, again favoring space over time.

4.2. Representing H as an array

Instead of the complex representation of H as described in Section 3.2, needed for an efficient support of the parent-operation, we used a simpler array-based approach: we mark the non-tree nodes in a bit-vector $P[0,n)$. (In the example of Fig. 2, we have $P = 00010011$ 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 P 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.

[5] We did not theoretically investigate codes that exploit the fact that the distribution of the 0's, 1's, and 2's in B is not necessarily uniform. However, in Section 5.2 we present a practical way achieving exactly this (basically a two-level wavelet tree [21] consisting of two 0-order compressed bit-vectors [38]).

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

n	Space [MByte]		Time for children [µsec]		Time for parents [µsec]	
	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	0.0337	0.3812	0.0138
100 000 000	188.1542	3654.4394	0.4095	–	0.4198	–

4.3. Using a wavelet tree for H

A different practical approach for representing $H[0,k)$ is the *wavelet tree* [21]. A wavelet tree on H is a binary tree with h leaves and is recursively constructed as follows: if the sequence consists of at least 2 different characters ($h \geq 2$), we split the alphabet into two halves: the first half consists of all characters $\leq h/2$, and the second of those characters $> h/2$. Then we construct a bit-vector $V[0,k)$ such that $V[i] = 0$ iff $H[i]$ belongs to the first half of the alphabet; this bit-vector V is stored at the root and partitions H into two subsequences, H_ℓ and H_r. The left and right children of the root are then the (recursively constructed) wavelet trees for H_ℓ and H_r. By adding rank- and select-support on all bit-vectors of the tree, the wavelet tree supports $\mathsf{select}_c(H,i)$ and $\mathsf{rank}_c(H,i)$ queries in $O(\lg h)$ time for every character c in S, while using $(k \lg h)(1 + o(1))$ bits.[6]

5. Practical results

We conducted three tests. A first test (Section 5.1) compares a basic implementation of our GLOUDS to a conventional adjacency-list based graph representation, using the example of phylogenetic networks. The second test (Section 5.2) on general tree-like graphs uses an even more space-conscious implementation of the GLOUDS, and (for fairness of comparison) also makes some space-optimization on the pointer-based representation. A final test compares our GLOUDS with a data structure for web graphs by performing breadth-first-traversals on real-world graphs.

Our machine was equipped with an Intel Core i7@2.2GHz and 8 GB of RAM, running under Windows 7. We compiled the program of Section 5.1 for 32 bits, in order not to make the pointer-based representation unnecessarily large. All programs used only a single core of the CPU.

5.1. Graphs representing phylogenetic networks

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 than trees. Quoting Huson and Scornavacca [24], 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 phylogenetic networks (no interchange of genetic material with extinct species).

For our first test, we constructed the GLOUDS as described in Section 4, and compared it 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). 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.

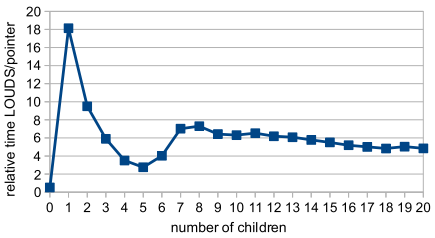
While there exist many implementations of succinct data structures for trees,[7] we are not aware of any implementations for graphs, hence we did not compare our data structure to others.

Table 1 shows the sizes of the data structures and the average running times for the children- and parents-operations with either representation.[8] We averaged the running times over 1000 tests for $n = 10\,000$, over 100 tests for $100\,000 \leq$

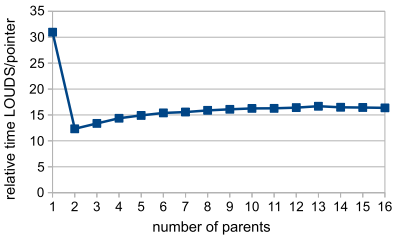
[6] Note that having rank-support on H is useful for an additional query on directed graphs, namely that of checking the presence of a (directed) edge: The edge (i, j) is present in G if and only if j "appears" between positions $a := \mathsf{select}_0(B,i)$ and $b := \mathsf{select}_0(B,i+1)$ in B. This appearance can be either as a tree edge (which is easily checked with rank_1-queries on B), or as a non-tree edge. The latter can be resolved by checking if or not $\mathsf{rank}_j(H, \mathsf{rank}_2(B,a)) \stackrel{?}{=} \mathsf{rank}_j(H, \mathsf{rank}_2(B,b))$. The total time for this operation is dominated by the time for rank-queries on H, which is $O(\log h)$ when using a wavelet tree.

[7] 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 [1,17].

[8] For memory reasons, the running times of the pointer-based representation could not be measured for the last 3 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.

$n \leq 1\,000\,000$, over 15 tests for $n = 10\,000\,000$, and over 5 tests for $n = 100\,000\,000$. 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 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. 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 Section 4.2 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.

5.2. General tree-like graphs

For our second test we used the succinct data structure library *sdsl*[9] to represent all data-structures, and created general graphs with no restrictions on the non-tree nodes (also allowing loops). We implemented and compared the performance of three data structures:

GLOUDS A slightly improved version of the implementation presented in Section 5.1. The trit-vector (as described in Section 4.1 is replaced by two bit-vectors: one of length $2n + k - 1$, where the `1`'s represent the trits `1` and `2`, and a second bit-vector of size $n + k$ to distinguish the `1`'s from the `2`'s (this is basically a two-level wavelet tree on the trit vector). Both bit-vectors are compressed with the technique by Raman et al. [38]. This representation turned out to be smaller than the one from Section 5.1, at no observable costs in running times for the operations.

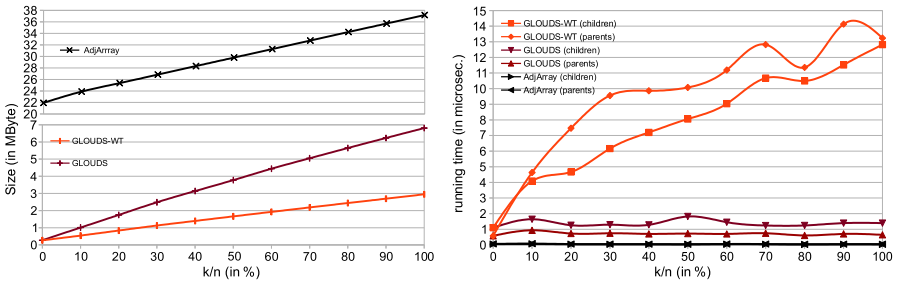
GLOUDS-WT The same as GLOUDS, but now H is stored as a wavelet tree (as described in Section 4.3).

AdjArray An adjacency-array based representation for static graphs [30, p. 168ff], tuned for space efficiency by using small pointers of size $\lceil \lg m \rceil$ bits using sdsl.

In the initial test phase we also evaluated a data-structure consisting of two RRR-compressed adjacency matrices [38], one for the children and one for the parents. This data structure was extremely big for tree-like graphs, even compared to the pointer-based representation (at least 70 times bigger) and also very slow (around 200 times slower than the GLOUDS representation), so it was not considered in further tests.

Fig. 4 shows the sizes and running times of all three implementations, with $n = 1\,000\,000$ nodes and values for k varying between 0 and n. It can be observed (a) that GLOUDS-WT is even smaller than GLOUDS, in particular for larger values

[9] https://github.com/simongog/sdsl.



(a) Sizes of the three representations.

(b) Running times to retrieve one child/parent.

Fig. 4. Detailed evaluation of the three representations.

of k. However, this comes at another increase in running time for the operations (b), by roughly one order of magnitude when compared with GLOUDS. (The AdjArray was again 10–20 times faster than GLOUDS.) Interestingly, the running time for GLOUDS-WT rises with increasing value of k, while those for GLOUDS and AdjArray stay more or less constant. This can naturally be explained by the increasing height of the wavelet tree for larger values of k (and hence also larger h).

5.3. Breadth first traversals on real-world graphs

The aim of this section is to show the practicality of our data structure by using it on real-world graphs, and applying a natural procedure on the resulting representation, namely a breadth-first-traversal (BFT). We compared the GLOUDS with a framework called WebGraph, which was especially designed to compress web graphs [11,10,9,8]. In these graphs the nodes represent web-pages, and the directed edges are the hyper-links. Several properties of web graphs have been identified and exploited to achieve compression:

- **Locality of reference**: Most of the links from a site point within the site. By lexicographical URL ordering, the outgoing links point to nodes whose position is close to that of the current node. Gap encoding techniques can then be used to encode the differences.
- **Similarity of adjacency lists**: Many outgoing links are shared from nodes close in URL lexicographical order. Hence, compression can be achieved by using references to the similar list, plus a list of edits. Thereby, long intervals of consecutive numbers are formed, which again can be easily compressed.
- **Skewed distribution**: The distribution of the in-degrees and out-degrees of a node is bound to a power law.

In [11], RePair [29] was used to get further compression. RePair is a phrase-based compressor that permits fast and local decompression. It consists of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are possible. Our GLOUDS representation was not changed at all to take advantage of these facts.

We used the following real world graphs from http://law.di.unimi.it for our comparison (see also Table 2 for some characteristics of these graphs):

amazon-2008 A symmetric graph describing similarity among books as reported by the Amazon store.
cnr-2000 A small crawl of the Italian CNR domain.
in-2004 A small crawl of the .in domain performed for the Nagaoka University of Technology.
uk-2002 This graph has been obtained from a 2002 crawl of the .uk domain performed by UbiCrawler.
enwiki-2013 This graph represents a snapshot of the English part of Wikipedia as of late February 2013. The pages are the nodes, and links between pages are the edges of the graph.

We also included a couple of *undirected graphs* to our test:

dblp-2010 DBLP is a bibliography service from which an undirected scientific collaboration network can be extracted: each vertex represents a scientist and two vertices are connected if they have worked together on an article. The DBLP database is from the year 2010.
dblp-2011 The DBLP database from the year 2011.
hollywood-2011 One of the most popular undirected social graphs: the graph of movie actors. Vertices are actors, and two actors are joined by an edge whenever they appeared in a movie together.

In cases where the graphs consist of more than root, we used Tarjan's algorithm [40] to identify the strongly connected components, from which the roots can be easily inferred.

Table 2
Number of nodes, edges, components and non-tree nodes in the graphs.

	Graph	Nodes	Edges	Components	Non-tree nodes
Directed	amazon-2008	735 323	5 158 388	1	4 423 066
	cnr-2000	325 557	3 216 152	1	2 890 596
	in-2004	1 382 908	16 917 053	211	14 420 349
	uk-2002	18 520 486	298 113 762	138 916	279 606 387
	enwiki-2013	4 206 785	101 355 853	367 984	97 089 663
Undirected	dblp-2010	326 186	1 615 400	74 724	515 966
	dblp-2011	986 324	6 707 236	174 519	2 463 041
	hollywood-2011	2 180 759	228 985 632	267 394	111 787 890

Table 3
Comparison between the WebGraph representation and the GLOUDS.

	Graph	Space [MByte]		Time for BFT [sec]		BFT per node [µsec]	
		GLOUDS	WebGraph	GLOUDS	WebGraph	GLOUDS	WebGraph
Directed	amazon-2008	8.551	13.654	0.95	0.86	1.29	1.17
	cnr-2000	3.069	2.765	0.36	0.63	1.12	1.95
	in-2004	15.813	11.231	1.60	1.59	1.16	1.15
	uk-2002	324.108	187.494	32.04	22.96	1.73	1.24
	enwiki-2013	208.896	322.257	5.58	12.57	1.32	2.99
Undirected	dblp-2010	0.992	1.752	0.36	0.54	1.11	1.68
	dblp-2011	5.348	8.501	1.18	1.26	1.20	1.28
	hollywood-2011	202.991	18.532	2.59	14.37	1.20	6.59

The advantage of the GLOUDS representation is that one can also efficiently navigate to the incoming edges, which the WebGraph framework cannot. Here, also the transposed graph needs to be saved. Additionally, our representation allows random access to every node, whereas the WebGraph framework in its most compressed form has only sequential access. Hence, we compared our structure to the bigger web-graphs with random access, adding the offsets and the size of the transposed graph.

Table 3 compares the GLOUDS with the WebGraph representation. As mentioned before, we added the sizes of the transposed graphs to the whole size in the WebGraph representation. This was not done for the graphs *dblp-2010*, *dblp-2011* and *hollywood-2011*, as these are undirected graphs. Still, the GLOUDS representation only needs to store half of the edges, because it can also traverse the incoming edges.

As the GLOUDS is already in level order, we can efficiently do a BFT by just traversing the tree edges starting from the super-root of our GLOUDS, because every non-tree edge points to a node that was already found. Still, if the traversal needs to be a BFT starting from an arbitrary node, the non-tree edges must be checked directly, by looking them up in the wavelet-tree which takes additional time. Depending on the number of non-tree edges h, we observed that the BFT can be slowed down by a factor of 15. For our tests we were interested in the minimum time a complete traversal took. Therefore, we started the BFT at the root node of the GLOUDS.

As the WebGraph framework was initially developed for Java, we conducted our BFT tests with the original Java implementation. Afterwards, we exported the graphs into a readable binary format for our C++ implementation, which then constructed the GLOUDS and started the BFT. We conducted the test, with the Java Framework and our C++ implementation, on the machine described at the beginning of this section.

It can be observed that graphs with few edges yield good results with the GLOUDS representation, in some cases they are even smaller: e.g., *dblp-2010* compresses to nearly half of the WebGraph representation. Nevertheless, graphs with many edges like *hollywood-2011* are more than 10 times bigger. Table 4 shows the sizes in proportion to each other, also looking at the quotient $\frac{m}{n}$. Additionally, it shows the sizes of the parts of which the GLOUDS consists of: the wavelet tree (of size sizeWT) for the non-tree edges, and the trit-vector (of size sizeTrit), which is actually made out of a second wavelet-tree compressing two bit-vectors, as mentioned at the beginning of Section 5.2.

To conclude this section, we think that our GLOUDS also compresses web-graphs really well, if $\frac{m}{n} < 10$, where m is the number of directed edges. For undirected graphs $\frac{m}{n} < 30$ seems to be a good estimation, where m still represents the number of directed edges, which are twice as many as the original undirected edges, one for every direction. For the graphs that are called "social graphs" at http://law.di.unimi.it, like *enwiki-2013*, the compression rate of the WebGraph framework is not that high, which makes the GLOUDS compression better even for $\frac{m}{n} \approx 24$. But as mentioned before, the BFT times rise with the number of non-tree edges, if the BFT is started from an arbitrary node. Hence, we still think that $\frac{m}{n} < 10$ is a good estimation for using the GLOUDS over the WebGraph framework. If in-degree node access is not needed $\frac{m}{n} < 5$ should be the limit.

Still, it should be noted that our GLOUDS could be further compressed for web-graphs, namely by reducing the wavelet tree size, which makes up for most of the space. This could be achieved by finding similarities, e.g., by using the RePair method on H. Hereby, out-neighbors could be listed by "unravelling" the blocks, and in-neighbors could be found by first

Table 4
Size comparison of the WebGraph representation and the GLOUDS.

	Graph	$\frac{m}{n}$	$\frac{\text{sizeGLOUDS}}{\text{sizeWebGraph}}$	sizeTrit	sizeWT
Directed	amazon-2008	7.015	0.626	0.802	7.749
	cnr-2000	9.879	1.109	0.287	2.782
	in-2004	12.232	1.408	1.302	14.511
	uk-2002	16.096	1.728	23.022	301.086
	enwiki-2013	24.093	0.648	7.006	201.890
Undirected	dblp-2010	4.952	0.566	0.171	0.821
	dblp-2011	6.800	0.628	0.639	4.699
	hollywood-2011	105.003	10.95	5.057	197.934

looking in which blocks the respective node lies in, and then by selects of the block-numbers on the initial vector, which would not be a trit-vector any more. This should lead to higher access times and higher creation time of the GLOUDS, in favor of reducing space for special graphs. As we focused our GLOUDS representation on arbitrary tree-like graphs, we leave this open to further research.

6. Conclusions

In this paper we presented a framework and implementation for a new succinct data structure for "tree-like" graphs based on the LOUDS representation for trees, which we called GLOUDS. We took a three-pronged approach to evaluate our representation. Firstly, we created random data structures inspired by phylogenetic networks with 10% of non-tree edges and compared the runtime of the operations children and parents on every node matched to a pointer based representation. The evaluation confirmed that our succinct data structure is practically feasible with a space reduction of around 95%. Secondly, a test on general graphs was performed by the use of a succinct library, also compressing the pointer-based data-structure. The practical evaluations again confirmed that the GLOUDS achieves a significant space reduction. Lastly, we showed how well our data-structure performs against a framework for web-graphs. The evaluation on "real-world" graphs shows that the GLOUDS performs well for graphs where $m < 10n$. In total, a trade-off between space and time can be observed, which is common in the world of succinct data structures.

Acknowledgements

We thank the anonymous reviewers for several helpful suggestions that improved the quality of this article.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proc. ALENEX, SIAM, 2010, pp. 84–97.
[2] J. Barbay, F. Claude, G. Navarro, Compact rich-functional binary relation representations, in: Proc. LATIN, in: LNCS, vol. 6034, Springer, 2010, pp. 170–183.
[3] D. Belazzougui, G. Navarro, New lower and upper bounds for representing sequences, in: Proc. ESA, in: LNCS, vol. 7501, Springer, 2012, pp. 181–192.
[4] D. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292.
[5] D.K. Blandford, G.E. Blelloch, I.A. Kash, Compact representations of separable graphs, in: Proc. SODA, ACM/SIAM, 2003, pp. 679–688.
[6] D.K. Blandford, G.E. Blelloch, I.A. Kash, An experimental analysis of a compact graph representation, in: ALENEX/ANALC, SIAM, 2004, pp. 49–61.
[7] A. Blumer, J. Blumer, D. Haussler, R.M. McConnell, A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis, J. ACM 34 (3) (1987) 578–595.
[8] P. Boldi, B. Codenotti, M. Santini, S. Vigna, Ubicrawler: a scalable fully distributed web crawler, Softw. Pract. Exp. 34 (8) (2004) 711–726.
[9] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks, in: Proceedings of the 20th International Conference on World Wide Web, ACM Press, 2011.
[10] P. Boldi, S. Vigna, The WebGraph framework I: compression techniques, in: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004), Manhattan, USA, ACM Press, 2004, pp. 595–601.
[11] F. Claude, G. Navarro, Fast and compact web graph representations, ACM Trans. Web 4 (4) (2010), Article No. 16.
[12] P. Davoodi, R. Raman, S.R. Satti, Succinct representations of binary trees for range minimum queries, in: Proc. COCOON, in: LNCS, Springer, 2012, pp. 396–407.
[13] A. Farzan, J. Fischer, Compact representation of posets, in: Proc. ISAAC, in: LNCS, vol. 7074, Springer, 2011, pp. 302–311.
[14] A. Farzan, J.I. Munro, Succinct representation of arbitrary graphs, in: Proc. ESA, in: LNCS, vol. 5193, Springer, 2008, pp. 393–404.
[15] A. Farzan, J.I. Munro, A uniform approach towards succinct representation of trees, in: Proc. SWAT, in: LNCS, vol. 5124, Springer, 2008, pp. 173–184.
[16] C. Gavoille, N. Hanusse, On compact encoding of pagenumber k graphs, Discret. Math. Theor. Comput. Sci. 10 (3) (2008) 23–34.
[17] R.F. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, Theor. Comput. Sci. 368 (3) (2006) 231–246.
[18] S. Gog, E. Ohlebusch, Fast and lightweight LCP-array construction algorithms, in: Proc. ALENEX, SIAM, 2011, pp. 25–34.
[19] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: Proc. SODA, ACM/SIAM, 2006, pp. 368–373.
[20] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA), CTI Press and Ellinika Grammata, Greece, 2005, pp. 27–38.
[21] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. SODA, ACM/SIAM, 2003, pp. 841–850.
[22] R. Grossi, G. Ottaviano, Design of practical succinct data structures for large data collections, in: Proc. SEA, in: LNCS, vol. 7933, Springer, 2013, pp. 5–17.
[23] Y. Gurevich, L. Stockmeyer, U. Vishkin, Solving NP-hard problems on graphs that are almost trees and an application to facility location problems, J. ACM 31 (3) (1984) 459–473.

[24] D.H. Huson, C. Scornavacca, A survey of combinatorial methods for phylogenetic networks, Genome Biol. Evol. 3 (23) (2011).
[25] G.J. Jacobson, Space-efficient static trees and graphs, in: Proc. FOCS, IEEE Computer Society, 1989, pp. 549–554.
[26] S. Joannou, R. Raman, Dynamizing succinct tree representations, in: Proc. SEA, in: LNCS, vol. 7276, Springer, 2012, pp. 224–235.
[27] S. Kannan, M. Naor, S. Rudich, Implicit representation of graphs, SIAM J. Discrete Math. 5 (4) (1992) 596–603.
[28] D.E. Knuth, Fundamental Algorithms, 3rd edition, The Art of Computer Programming, vol. 1, Addison Wesley, 1997.
[29] N.J. Larsson, A. Moffat, Offline dictionary-based compression, in: Proc. DCC, IEEE Press, 1999, pp. 296–305.
[30] K. Mehlhorn, P. Sanders, Algorithms and Data Structures: The Basic Toolbox, Springer, 2008.
[31] J.I. Munro, Tables, in: Proc. FSTTCS, in: LNCS, vol. 1180, Springer, 1996, pp. 37–42.
[32] J.I. Munro, P.K. Nicholson, Compressed representations of graphs, in: M. Kao (Ed.), Encyclopedia of Algorithms, Springer, 2015.
[33] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations, in: Proc. ICALP, in: LNCS, vol. 2719, Springer, 2003, pp. 345–356.
[34] J.I. Munro, V. Raman, Succinct representation of balanced parentheses, static trees and planar graphs, in: Proc. FOCS, IEEE Computer Society, 1997, pp. 118–126.
[35] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (3) (2001) 762–776.
[36] R. Pagh, Low redundancy in static dictionaries with constant query time, SIAM J. Comput. 31 (2) (2001) 353–363.
[37] M. Pătraşcu, Succincter, in: Proc. FOCS, IEEE Computer Society, 2008, pp. 305–313.
[38] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, ACM Trans. Algorithms 3 (4) (2007), Article No. 43.
[39] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[40] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.