


Breadth-First Rank/Select in Succinct Trees and Distance Oracles for Interval Graphs

Meng He

Dalhousie University, Canada
mhe@cs.dal.ca

J. Ian Munro

University of Waterloo, Canada
imunro@uwaterloo.ca
 <https://orcid.org/0000-0002-7165-7988>


Yakov Nekrich

Michigan Tech, USA
yakov@mtu.edu

Sebastian Wild

University of Liverpool, UK
wild@liverpool.ac.uk
 <https://orcid.org/0000-0002-6061-9177>

Kaiyu Wu

University of Waterloo, Canada
k29wu@uwaterloo.ca
 <https://orcid.org/0000-0001-7562-1336>

Abstract

We present the first succinct data structure for ordinal trees that supports the mapping between the preorder (i.e., depth-first) ranks and level-order (breadth-first) ranks of nodes in constant time. It also provides constant-time support for all the operations provided by different approaches in previous work, as well as new operations that allow us to retrieve the last internal node before or the first internal node after a given node in a level-order traversal. This new representation gives us the functionality needed to design the first succinct distance oracles for interval graphs, proper interval graphs and k -proper/ k -improper interval graphs.

2012 ACM Subject Classification Theory of computation \rightarrow Data structures design and analysis; Theory of computation \rightarrow Data compression

Keywords and phrases succinct data structures, ordinal tree, succinct tree representation, level order, breadth-first order, interval graph, k -proper interval graph, k -improper interval graph, proper interval graph, distance oracle, succinct graph representation

1 Introduction

As fundamental objects in computer science, trees are widely used in theory and applications. The standard pointer-based representation of trees uses $O(n)$ words or $O(n \log n)$ bits to represent trees on n nodes. As a result of the rapid growth of electronic data sets nowadays, this space costs can be a bottleneck for tree-based data structures to fit into faster levels of the memory hierarchy in computer systems. Thus, extensive work has been done to represent a tree *succinctly* [19, 11, 23, 24, 9, 25, 21, 33, 29, 6, 20, 4, 17, 18, 15], i.e., to design a data structure of $2n + o(n)$ bits to represent a tree while supporting navigational operations in constant time (on the word-RAM). Most works on succinct data structures for trees have

focused on *ordinal* trees, i.e., trees with unbounded degree where the order of children matters, but no distinction is made, e.g., between a left and a right single child. Some ideas have been translated to *cardinal* trees (and binary trees as a special case) [14, 12].

The *level-order unary degree sequence* (LOUDS) representation of an ordinal tree [19] consists of listing the degrees of nodes in unary encoding while traversing the tree with a breadth-first search. This is a direct generalization of the representation of heaps, i.e., complete binary trees stored in an array in breadth-first order: There, due to the completeness of the tree, no extra information is needed to map the rank of a node in the breadth-first traversal to the ranks of its parent and children in the tree. The LOUDS is exactly the required information to do the same for general ordinal trees. Historically one of the first schemes to succinctly represent a static tree, LOUDS is still liked for its simplicity and practical efficiency [2], but a major disadvantage of LOUDS-based data structures is that they support only a very limited set of operations [28].

Replacing the breadth-first traversal by a depth-first traversal yields the *depth-first unary degree sequence* (DFUDS) encoding of a tree, based on which succinct data structures with efficient support for many more operation have been designed [6]. Other approaches that allow to support largely the same set of operations are based on the *balanced-parentheses* (BP) encoding [23] or rely on *tree covering* (TC) [17] for a hierarchical tree decomposition.

One operation that has gained some notoriety for not being supported by any of these approaches, (DFUDS, BP, or TC), is computing the rank of a node (or selecting a node by its rank) in the breadth-first traversal of the tree. In particular, no succinct tree data structure was known that could efficiently support the mapping between preorder (i.e., depth-first) ranks and level-order (breadth-first) ranks of nodes.

In this paper, we present such a data structure based on TC. This solves an open problem of [18], and provides further evidence for the flexibility of the tree-covering approach.

Supporting level-order ranks is not an arbitrary *etude* in adding another esoteric operation to the long list of operations for succinct trees – on the contrary, it was the missing keystone for succinct distance oracles for interval graphs, proper interval graphs and k -proper/improper interval graphs; these data structures are the second contribution of this article. Interval graphs have applications in operational research [3] and bioinformatics [37], and designing distance oracles is a fundamental problem interesting in its own right [38, 35, 30].

Our tree data structure is based on a novel way to (recursively) decompose a tree into *forests* of subtrees in a way that makes storing level-wise information affordable. Although most constructions to support operations could be adapted to work with the new partitioning scheme, some required substantial modifications.

Our Results on Trees. Our first result is a succinct representation of ordinal trees which occupies $2n + o(n)$ bits and supports all operations listed in Table 1 in $O(1)$ time, that is, all operations supported by previous work plus these new operations:

- `node_rankLEVEL(v)` and `node_selectLEVEL(i)`: computing the position of node v in a level-order traversal of the tree resp. finding the i th node in the level-order traversal;
- `prev_internal(v)` and `next_internal(v)`: the non-leaf node closest in level-order to v that comes before resp. after v in level order.

Previously, `node_rankLEVEL` and `node_selectLEVEL` were only supported by the LOUDS representation of trees [19], which, however, does not support rank/select by preorder (and generally only supports a limited set of operations). Hence our trees are the only succinct data structures to map between preorder (i.e., depth-first) ranks and level-order (breadth-first)

<code>parent(v)</code>	the parent of v , same as <code>anc(v, 1)</code>
<code>degree(v)</code>	the number of children of v
<code>child(v, i)</code>	the i th child of node v ($i \in \{1, \dots, \text{degree}(v)\}$)
<code>child_rank(v)</code>	the number of siblings to the left of node v plus 1
<code>depth(v)</code>	the depth of v , i.e., the number of edges between the root and v
<code>anc(v, i)</code>	the ancestor of node v at depth <code>depth(v) - i</code>
<code>nbdesc(v)</code>	the number of descendants of v
<code>height(v)</code>	the height of the subtree rooted at node v
<code>LCA(v, u)</code>	the lowest common ancestor of nodes u and v
<code>leftmost_leaf(v)</code>	the leftmost leaf descendant of v
<code>rightmost_leaf(v)</code>	the rightmost leaf descendant of v
<code>level_leftmost(ℓ)</code>	the leftmost node on level ℓ
<code>level_rightmost(ℓ)</code>	the rightmost node on level ℓ
<code>level_pred(v)</code>	the node immediately to the left of v on the same level
<code>level_succ(v)</code>	the node immediately to the right of v on the same level
<code>prev_internal(v)</code>	the last internal node before v in a level-order traversal
<code>next_internal(v)</code>	the first internal node after v in a level-order traversal
<code>node_rank$_X$(v)</code>	the position of v in the X -order, $X \in \{\text{PRE}, \text{POST}, \text{IN}, \text{DFUDS}, \text{LEVEL}\}$, i.e., in a preorder, postorder, inorder, DFUDS order, or level-order traversal of the tree
<code>node_select$_X$(i)</code>	the i th node in the X -order, $X \in \{\text{PRE}, \text{POST}, \text{IN}, \text{DFUDS}, \text{LEVEL}\}$
<code>leaf_rank(v)</code>	the number of leaves before and including v in preorder
<code>leaf_select(i)</code>	the i th leaf in preorder

■ **Table 1** Navigational operations on succinct ordinal trees. (v denotes a node and i an integer).

ranks in constant time. We list concrete applications below, but expect this to be a vital building block for future work. Table 2 in Appendix A compares our result to previous work.

Our Results on Interval Graphs. Interval graphs are intersection graphs of intervals on the line; several subclasses are obtained by further restricting how the intervals can intersect: no interval is properly contained in another (*proper interval graphs*), or every interval is contained by (contains) at most k other intervals (*k -proper* resp. *k -improper interval graphs*). The problem of representing these graphs succinctly has been studied by Acan et al. [1], but without efficient distance queries. We present succinct representations of interval graphs, proper interval graphs and k -proper/ k -improper graphs in $n \log n + (5 + \varepsilon)n + o(n)$, $2n + o(n)$ and $2n \log k + 8n + o(n \log k)$ bits, respectively, where n is the number of vertices and ε is an arbitrarily small constant, such that the following operations are supported:

- `degree(v)`: the degree of v , i.e., the number of vertices adjacent to v in $O(1)$ time;
- `adjacent(u , v)`: whether vertices u and v are adjacent in $O(1)$ time;
- `neighborhood(v)`: iterating through the vertices adjacent to v in $O(1)$ time per neighbor;
- `spath(u , v)`: listing a shortest path from vertex u to v in $O(|\text{spath}(u, v)|)$ time;
- `distance(u , v)`: the length of the shortest path from u to v in $O(1)$ time.

This is the query that a distance oracle is expected to answer and which was not efficiently supported in [1]. Our implementation is made possible by level-order rank/select on trees.

The succinctness of our representation is evidenced by the information-theoretic lower bounds of $n \log n - 2n \log \log n - O(n)$ bits [1] and $2n - O(\lg n)$ bits [16] on representing interval graphs and proper interval graphs, respectively.

2 Related Work

Succinct Representations of Ordinal Trees. The LOUDS representation, first proposed by Jacobson [19] and later studied by Clark and Munro [11] under the word RAM, uses $2n + o(n)$

bits to represent a tree on n nodes, such that, given a node, its first child, next sibling and parent can be located in constant time. Three other approaches, BP, DFUDS or TC, have since been proposed to support more operations while still using $2n + o(n)$ bits.

As the oldest tree representation after LOUDS, BP-based representations have seen a long history of successive improvements and uses in various applications of succinct trees. The list of supported operations has grown over a sequence of several works [23, 24, 9, 25, 21, 33, 29] to include all standard operations, bar the level-order ones and `node_rankDFUDS` / `node_selectDFUDS`.

The other representations have a similar history, albeit shorter, and we refer to [6, 20, 4] for DFUDS and [17, 18, 15] for TC. A full survey is also given in Appendix A. Table 2 summarizes the operations supported by each of these three approaches. Other than supporting more operations, work has done for alternative goals such as achieving compression [20, 14], reducing redundancy [29] and supporting updates [29].

Succinct Representations of Interval Graphs. Several succinct representations of (subclasses of) graphs have been studied, e.g., for general graphs [13], k -page graphs [19], certain classes of planar graphs [10, 9, 8], separable graphs [7], posets [22] and distributive lattices [26]. Recently, Acan et al. [1] showed how to represent an *interval graph* on n vertices in $n \log n + (3 + \varepsilon)n + o(n)$ bits to support `degree` and `adjacent` in $O(1)$ time, `neighborhood`(v) in $O(\text{degree}(v))$ time and `spath`(u, v) in $O(|\text{spath}(u, v)|)$ time, where ε is a positive constant that can be arbitrarily small. To show the succinctness of their solution, they proved that $n \log n - 2n \log \log n - O(n)$ bits are necessary to represent an interval graph. They also showed how to represent a *proper interval graph* and a *k-proper/k-improper interval graph* in $2n + o(n)$ and $2n \log k + 6n + o(n \log k)$ bits, respectively, supporting the same queries.

Distance Oracles over Chordal Graphs and Interval Graphs. Exact and approximate distance oracles for graphs are well-studied [38, 35, 30], and we only focus on the works closest to our results. Ravi et al. [32] consider the problem of solving the all-pair shortest path problem over interval graphs in optimal $O(n^2)$ time in 1992. Later, Singh et al. [34] designed a data structure of $O(n)$ words that can *approximate* the distance between two vertices u and v in a chordal graph in $O(1)$ time, and the answer is between $|\text{distance}(u, v)|$ and $2|\text{distance}(u, v)| + 8$. (Interval graphs are a subclass of chordal graphs, so their result applies to the former, as well.) More recently, Munro and Wu [27] designed a succinct representation of chordal graphs using $n^2/4 + o(n^2)$ bits, which inspired our new distance oracles. Their representation offers an (albeit slow) *exact* distance oracle (supporting `distance` in $O(nf(n))$ time for any $f(n) \in \omega(1)$), and supports `adjacent` in $O(f(n))$ time, `degree` in $O(1)$ time and `neighborhood` in $O(f^2(n))$ time per neighbor. The space cost matches the information-theoretic lower bound of representing a chordal graph [36] up to lower order terms. They also designed an *approximate* distance oracle of $n \log n + o(n \log n)$ bits with $O(1)$ query time, where answers are within 1 of the actual distance.

3 Notation and Preliminaries

We write $[n..m] = \{n, \dots, m\}$ and $[n] = [1..n]$ for integers n, m . We use \lg for \log_2 and leave the basis of \log undefined (but constant); (any occurrence of \log outside an Landau-term should thus be considered a mistake). As is standard in the field, all running times assume the word-RAM model with word size $\Theta(\log n)$.

We use the data structure of Raman, Raman, and Rao [31] for compressed bitvectors:

► **Lemma 1** (Compressed bit vector). *Let \mathcal{B} be a bit vector of length n , containing m 1-bits. There is a data structure using $\lg \binom{n}{m} + O\left(\frac{n \log \log n}{\log n}\right) \leq m \lg \left(\frac{n}{m}\right) + O\left(\frac{n \log \log n}{\log n} + m\right)$ bits of space that supports the following operations in $O(1)$ time, for any $i \in [1, n]$:*

- $\text{access}(\mathcal{V}, i)$: return the bit at index i in \mathcal{V} .
- $\text{rank}_\alpha(\mathcal{V}, i)$: return the number of bits with value $\alpha \in \{0, 1\}$ in $\mathcal{V}[1..i]$.
- $\text{select}_\alpha(\mathcal{V}, i)$: return the index of the i -th bit with value $\alpha \in \{0, 1\}$.

4 Tree Slabbing

In this section, we describe the new tree-covering method used in our data structure. Throughout this paper, let T be an ordinal tree over n nodes. We will identify nodes with their ranks $1, \dots, n$ (order of appearance) in a preorder traversal. Tree covering (TC) relies on a two-tier decomposition: a tree consists of mini trees, which each consists of micro trees. The former will be denoted by μ^i , the latter by μ_j^i .

4.1 The Farzan-Munro Algorithm

We will build upon previously used tree covering schemes. A greedy bottom-up approach suffices to break a tree of n nodes into $O(n/B)$ subtrees of $O(B)$ nodes each [17]. However, more carefully designed procedures yield restrictions on the touching points of subtrees:

► **Lemma 2** (Tree Covering, [14, Thm. 1]). *For any parameter $B \geq 3$, an ordinal tree with n nodes can be decomposed, in linear time, into connected subtrees with the following properties.*

- Subtrees are pairwise disjoint except for (potentially) sharing a common subtree root.
- Each subtree contain at most $2B$ nodes.
- The overall number of subtrees is $\Theta(n/B)$.
- Apart from edges leaving the subtree root, at most one other edge leads to a node outside of this subtree. This edge is called the “external edge” of the subtree.

By inspection of the proof, we can say a bit more: If v is a node in the (entire) tree and is also the root of several subtrees (in the decomposition), then the way that v ’s children (in the entire tree) are divided among the subtrees is into *consecutive* blocks. Each subtree contains at most two of these blocks. (This case arises when the subtree root has exactly one heavy child in the decomposition algorithm.)

Why is level-order rank/select hard? Suppose we try to compute the level-order rank of a node v , and we try to reduce the global query (on the entire tree T) to a local query that is constrained to a mini tree μ^i . This task is easy if we can afford to store the level-order ranks of the leftmost node in μ^i for each level of μ^i : then the level-order rank of v is simply the global level-order rank of w , the leftmost node in μ^i on v ’s level (v ’s depth), plus the local level-order rank of v , minus the local level-order rank of w minus one (since we double counted the nodes in μ^i on the levels above w).

However, for general trees, we cannot afford to store the level-order rank of all leftmost nodes. This would require $\text{height}(\mu^i) \cdot \lg n$ bits for $\text{height}(\mu^i)$ the height of μ^i ; towards a sublinear overhead in total, we would need a $o(1)$ overhead per node, which would (on average) require μ^i to have $|\mu^i| = \omega(\text{height}(\mu^i) \log n)$ nodes or $\text{height}(\mu^i) = o(|\mu^i| / \log n)$. Since the tree T to store can be one long path (or a collection of few paths with small off-path subtrees etc.), any approach based on decomposing T into induced subtrees is bound to fail the above requirement.

The solution to this dilemma is the observation that the above “bad trees” have another feature that we can exploit: The total number of nodes on a certain interval of levels is small. If we keep such an entire horizontal slab of T together, translating global level-order rank queries into local ones does not need the ranks of all leftmost nodes: everything in these levels is entirely contained in μ^i now, and it suffices to add the level-order rank of the (leftmost) root in μ^i .

Our scheme is essentially based on decomposing the tree into parts that are one of these two extreme cases – “skinny slabs” or “fat subtrees” – and counting them separately to amortize the cost for storing level-order information.

4.2 Covering by Slabs

In order to control the height of the subtrees in Lemma 2 we will perform a preprocessing step. We fix two parameters: $H \in \mathbb{N}$, the height of slabs, and $B > H$, the target block size. We start by cutting T horizontally into slabs of thickness/height exactly H , but we allow ourselves to start cutting at an offset $o \in [H]$. That means that the topmost and bottommost slabs might have height $< H$. We choose o so as to minimize the total number of nodes on levels at which we make the horizontal cuts. We call these nodes *s-nodes* (“slabbed nodes”), and their parent edges *slabbed edges*. A simple counting argument shows that the number of *s-nodes* (and slabbed edges) is at most n/H .

We will identify induced subgraphs with the set of nodes that they are induced by. So $S_i = \{v : \text{depth}(v) \in [(i-1)H + o .. iH + o]\}$, the set of nodes making up the i th slab, also denotes the i th slab itself, $i = 0, \dots, h$. Obviously, the number of slabs is $h + 1 \leq n/H + 2$.

We note that the *s-nodes* are contained in *two* slabs. For any given slab, we will refer to the first *s-level* included as (original) *s-nodes* and the second as *promoted s-nodes*. We note that the first slab does not contain any *s-nodes* and the last slab does not contain any promoted *s-nodes*.

Recall that S_i is (in general) a set of subtrees, ordered by the left-to-right order of their roots; we will add a *dummy root* to turn it into a single tree. We note that the *s-nodes* are the first (after the dummy root) and the last levels of any slab.

If $|S_i| \leq B$, S_i is a “skinny slab” and we will not further subdivide it. We will call these *skinny* subtrees (after adding the dummy root). If $|S_i| > B$, we apply the Farzan-Munro tree-covering scheme (Lemma 2) with parameter B to the slab (with the dummy root added) to obtain *fat subtrees*. This directly yields the following result.

► **Theorem 3 (Tree Slabbing).** *For any parameters $B > H \geq 3$, an ordinal tree T with n nodes can be decomposed, in linear time, into connected subtrees with the following properties.*

1. *Subtrees are pairwise disjoint except for (potentially) sharing a common subtree root.*
2. *Subtrees have size $\leq M = 2B$ and height $\leq H$.*
3. *Every subtree is either pure (a connected induced subgraph of T), or glued (a dummy root, whose children are connected induced subgraphs of T).*
4. *Every subtree is either a skinny (slab) subtree (an entire slab) or fat.*
5. *The overall number of subtrees is $O(n/H)$, among which $O(n/B)$ are fat.*
6. *Connections between subtrees μ and μ' are of the following types:*
 - a. *μ and μ' share a common root. Each subtree contains at most two blocks of consecutive children of a shared root.*
 - b. *The root of μ' is a child of the root of μ .*
 - c. *The root of μ' is a child of another node in μ . This can happen for at most one node in μ .*

- d. μ' contains the original copy of a promoted s -node in μ . The total number of these connections is $O(n/H)$.

„Oans, zwoa, G'suffa“ The above tree-slabbing scheme has two parameters, H and B . As is typical for succinct data structures, we will invoke it *twice*, first using $H = \lceil \lg^3 n \rceil$ and $B = \lceil \lg^5 n \rceil$ to form m mini trees μ^1, \dots, μ^m of at most $M = 2B$ nodes each. While in general we only know $m = O(n/H) = O(n/\log^3 n)$, only $O(n/M) = O(n/\log^5 n)$ of these mini trees are *fat* subtrees (subtrees of a fat slab), the others being *skinny*.

Mini trees μ^i are recursively decomposed by tree slabbing with height $H' = \lceil \frac{\lg n}{(\lg \lg n)^2} \rceil$ and block size $B' = \lceil \frac{1}{8} \lg n \rceil$ into micro trees $\mu_1^i, \dots, \mu_{m_i}^i$ of size at most $M' = 2b = \frac{1}{4} \lg n$. The total number of micro trees $m' = m'_1 + \dots + m'_m = O(n/H')$, but at most $O(n/B')$ are fat micro trees. We note that after these two levels of recursion we have reached a size for micro trees small enough to use a “Four-Russian” lookup table (including support for various micro-tree-local operations) that takes sublinear space. We will refer to the s -nodes created at mini resp. micro tree level as *tier-1* resp. *tier-2* s -nodes.

Internal node ids Internally to our data structure, we will encode a node v by a tuple specifying the mini tree, the micro tree within the mini tree, and the node within micro tree. In the context of tree covering, these ids have been referred to as τ -names, and we will follow this tradition. More specifically, $\tau(v) = \langle \tau_1, \tau_2, \tau_3 \rangle$ means that v is the τ_3 th node in $\mu_{\tau_2}^{\tau_1}$, where nodes in a micro tree are numbers by the micro-tree-local preorder (DFS order) rank and mini/micro trees are likewise ordered by when their first node (the root) appears in a preorder traversal of T , ties (among subtrees sharing roots) broken by the second node.

We note that there are $O(n/H)$ mini trees, $O(B/H')$ micro trees inside one mini tree and $O(B')$ nodes in each micro tree; we can thus write down τ names with asymptotically $\sim \lg n + 2 \lg \lg n + 2 \lg \lg \lg n$ bits each. The concatenation $\tau_1(v)\tau_2(v)\tau_3(v)$ can be seen as a binary number; listing nodes in increasing order of that number gives the τ -order.

Who gets promotion? A challenge in tree covering is to handle operations like **child** when they cross subtree boundaries. The solution is to add the endpoint of a crossing edge also to the parent mini/micro tree; these copies of nodes are called (*tier-1/tier-2*) *promoted nodes*. They have their own τ -name, but actually refer to the same original node; we call the τ -name of the original node the *canonical τ -name*. A major simplification in [14] came from having all crossing edges (except for one) emanate from the subtree root, which allows us to implicitly represent these edges instead of promoting their endpoints. Only the external edge of a subtree requires promoting the endpoint to the subtree.

For tree slabbing, we additionally have slabbed edges to handle. As mentioned earlier, we promote all endpoints of slabbed edges into the parent slab *before we further decompose a slab*. That way, the size bounds for subtrees already include any promoted copies, but we blow up the number of subtrees by an – asymptotically negligible – factor of $1 + 1/H \sim 1$. Promoted s -nodes again have both canonical and secondary τ -names.

5 Operations

We now describe a list of operations that are needed for our application; many more are possible as sketched in Appendix B. We start describing some common concepts.

The *type* of a micro tree is the concatenation of its size (in Elias code), the BP of its local shape, and the preorder rank of the promoted dummy node (0 if there is none), and several

bits indicating whether the lowest level are promoted s -nodes, and whether the root is a dummy root. We store a variable-cell array of the *types* of all micro trees in τ -order. The BP of all micro trees will sum to $2(n + O(n/H')) = 2n + o(n)$ bits of space; the other components of the type are asymptotically negligible. Type consists of at most $\sim \frac{1}{2} \lg n$ bits, so we can store a table of all possible types with various additional precomputed local operations in $O(\sqrt{n} \text{polylog}(n))$ bits.

5.1 Preorder rank/select

We first consider how to convert between global preorder ranks and τ -names. Let us fix one level of subtrees, say mini trees. Consider the sequence $\tau_1(v)$ for all the nodes v in a preorder traversal. A node v so that $\tau_1(v) \neq \tau_1(v-1)$ is called a (*tier-1*) *preorder changer* [18, Def. 4.1]. Similarly, nodes v with $\tau_2(v) \neq \tau_2(v-1)$ are called (*tier-2*) *preorder changers*. We will associate with each node v “its” tier-1 (tier-2) preorder changer u , which is the last preorder changer preceding v in preorder, i.e., $\max\{u \in [1..v] : \tau_1(u) \neq \tau_1(u-1)\}$; (Recall that we identify nodes with their preorder rank.)

By Lemma 3, the number of tier-1 preorder changers is $O(n/H)$, since the only times a mini-tree can be broken up is through the external edge (once per tree), the two different blocks of children of the root, or at slabbed edges. Similarly, we have $O(n/H')$ tier-2 preorder changers. We can thus store a compressed bitvector (Lemma 1) to indicate which nodes in a preorder traversal are (tier-1/tier-2) preorder changers. The space for that is $O(\frac{n}{H} \log(H) + n \frac{\log \log n}{\log n}) = o(n)$ for tier 1 and $O(\frac{n}{H'} \log(H') + n \frac{\log \log n}{\log n}) = O(n \frac{(\log \log n)^3}{\log n}) = o(n)$ for tier 2.

We will additionally store a compressed bitvector indicating preorder changers by τ -name, i.e., we traverse all nodes in τ -order and add a 1 if the current node is a preorder changer, and a 0 if not. Again, we can afford to this using Lemma 1 for tier-1 and tier-2 in the same space as above. (The universe grows to $n \text{polylog}(n)$, but that does not affect the space by more than a constant factor).

In the same space, we can store $O(\log n)$ bits for each tier-1 changer and $O(\log \log n)$ bits for each tier-2 changer, and with above bitvectors, we can access that information given global preorder or τ -names.

Select Given the preorder number of a node v , we want to find $\tau(v)$. Let u and u' be the tier-1 resp. tier-2 preorder changers associated with v . The core observation is that $\tau_1(u) = \tau_1(v)$ and $\tau_2(u') = \tau_2(v)$, since a nodes tier-1 (tier-2) preorder changer by definition lies in the same mini (micro) tree as v . We thus store the array of τ_1 -numbers of all tier-1 preorder changers as they are visited by a preorder traversal of T . Using rank and select on the bitvectors from above, we find u , for which we look up τ_1 . The procedure applies, *mutatis mutandis*, to τ_2 using the tier-2 preorder changer u' . Since τ_2 is local to a mini tree, $\lg M = O(\log \log n)$ bits suffice, so we can afford to store τ_2 for every tier-2 changer. We also store the τ_3 -number for each tier-2 changer in the same space. We can then obtain $\tau_3(v)$ as the sum $\tau_3(u')$ and the distance from the last 1 in the bit vector indicating τ_2 changers.

Rank For space purposes, see Appendix B for the full description.

5.2 Level-order rank/select

Let w_1, \dots, w_n be the nodes of T in level order, i.e., w_i is the i th node visited in the left-to-right breadth-first traversal of T . Similar to the preorder, we call a node w_i a *tier-1* (*tier-2*)

level-order changer if w_{i-1} and w_i are in different mini (micro) trees. The following lemma bounds the number of tier-1 (tier-2) level-order changers (see Appendix D for its proof).

► **Lemma 4.** *The number of tier-1 (tier-2) level-order changers is $O(n/H + nH/B) = O(n/\log^2 n)$ ($O(n/H' + nH'/B') = O(n/(\log \log n)^2)$).*

With that preparation done, we proceed similarly as for preorder.

Select Given the level-order rank i , find $\tau(w_i)$. We store $\tau_1(w_1), \dots, \tau_1(w_n)$ in a piece-wise constant array, using the same technique as for preorder (compressed bitvector for changers, explicit values at changers), and similarly for $\tau_2(w_1), \dots, \tau_3(w_n)$. Both require $o(n)$ bits.

For τ_3 , we have to take an extra step as we don't visit nodes in preorder now. But we can store the micro-tree-local *level-order* rank at all tier-2 level-order changers and compute the distance of w_i from its tier-2 changer. The sum is the micro-tree-local level-order rank of w_i , which we translate to $\tau_3(w_i)$ using the lookup table.

Rank Given a node v by τ -name, we now seek the i with $v = w_i$. We proceed similarly as for preorder rank, i.e., we compute i as $j + (j' - j) + (i - j)$ for w_j and $w_{j'}$ the tier-1 resp. tier-2 level-order changers of $v = w_i$.

From the micro-tree lookup table, we obtain τ_3 of $w_{j'}$ and the level-order distance to v . For tier-2 changers, we store the mapping from τ to distance (in level order) to tier-1 changer, as well as $\langle \tau_2, \tau_3 \rangle$ of its tier-1 changer. Finally, for tier-1 changers, we map τ to its level-order rank. That determines all summands for i .

5.3 Local navigation

Various local navigation operations are possible; For the succinct distance oracle for interval graphs, we only need `last_child` and `parent`. Note that `parent(v) = anc(v, 1)`, so it is subsumed by the level-ancestor solution below. `last_child` can easily be implemented directly (details omitted); it can also be obtained as `last_child(v) = child(v, degree(v))`, see Appendix B.

5.4 Previous Internal in Level Order

Given $\tau(v)$, find `prev_internal(v) = $\tau(w)$` , where w is the last non-leaf node (`degree(w) > 0`) preceding v in level order. In the micro-tree lookup table, we store whether there is an internal node to the left of v inside the micro-tree, and if so, its τ_3 . If w does not lie in $\mu_{\tau_2(v)}^{\tau_1(v)}$, we get v 's tier-2 level-order changer u' from the lookup table, for which we store whether there is an internal node to the left of u' inside the micro-tree, and if so, store its $\langle \tau_2, \tau_3 \rangle$. If w is also not in $\mu^{\tau_1(v)}$, we move to u' 's tier-1 level-order changer ($\langle \tau_2(u), \tau_3(u) \rangle$ is stored for u'). At tier-1 changers u , we store `prev_internal(u)` directly.

5.5 Depth and Level-Ancestor

Depth Given $\tau(v)$, compute the level on which v lies. We store the global depth of the mini-tree root and the mini-tree-local depth at each micro-tree root. For a node v , find the depth relative to the micro-tree root using the lookup table, and add the mini-tree-local depth and the global depth. We may need to adjust for dummy roots but that is trivial.

Level-Ancestor Given $\tau(v)$, find $\text{anc}(v, i) = \tau(w)$ for w the ancestor of v on level $\text{depth}(v) - i$. The solution of [17, §3] essentially works without changes, but tree slabbing actually simplifies it slightly. We start by bootstrapping from a non-succinct solution for the level-ancestor (LA) problem:

► **Lemma 5** (Level ancestors, [5, Thm. 13]). *There is a data structure using $O(n \log n)$ bits of space that answers $\text{anc}(v, i)$ queries on a tree of n nodes in $O(1)$ time.*

Geary et al. apply this to a so-called macro tree; we observe that we can instead build the LA data structure for all tier-1 s-nodes, where s-nodes u and v are connected by a macro edge if there is a path from u to v in T that does not contain further s-nodes. This uses $O(\frac{n}{H} \log(\frac{n}{H})) = O(n/\log n)$ bits. Each mini-tree root stores its closest ancestor that is a tier-1 s-node. Additionally, mini/micro tree roots and (tier-1/tier-2) s-nodes store collections of *jump pointers*: mini trees / tier-1 s-nodes allow to jump to an ancestor at any distance in $1, 2, \dots, \sqrt{H}$ or $\sqrt{H}, 2\sqrt{H}, 3\sqrt{H}, \dots, H$; the same holds for micro trees / tier-2 s-nodes with H' instead of H , and as usual storing only $\langle \tau_2, \tau_3 \rangle$. (Mini-tree roots / tier-1 s-nodes store full τ -names in jump pointers.)

The query now works as follows (essentially [17, Fig. 6], but with care for s-nodes): We compute the micro-tree local depth of v by table lookup and check if w lies inside the micro tree; if so, we find it by table lookup. If not, we move to the micro-tree root – or the tier-2 s-node in case the micro-tree root is a dummy root (using a micro-tree local **anc** query); let's call this node x . We now compute x 's mini-tree local depth (using the data structures for **depth**) to check if w lies inside this mini-tree. If it does, we use x 's jump pointers: either directly to w (if the distance was at most $\sqrt{H'}$), or to get within distance $\sqrt{H'}$, from where we continue recursively. If w is not within the current mini-tree, we jump to y , the mini-tree root, or a tier-1 s-node in case the mini tree has a dummy root (using a recursive, mini-tree local **anc** query). If w is within distance H from there, we use y 's jump pointers (to either get to y directly, or to get within distance \sqrt{H}). Otherwise, we use y 's pointer to its next tier-1 s-node ancestor (unless y already is such). The LA data structure on tier-1 s-nodes allows us to jump within distance H of w , from where we continue.

Note that after following two root jump pointers of each kind we are always close enough to w that the next micro-tree root will have a direct jump pointer to w . The recursive call to find a tier-1 s-node subforest root (when a mini-tree has a dummy root) is always resolved local to the mini tree, so cannot lead to another such recursive calls. Hence the running time is $O(1)$.

Combing our work in Sections 4, 5 and Appendix B, we have our first result:

► **Theorem 6** (Succinct trees). *An ordinal tree on n nodes can be represented in $2n + o(n)$ bits to support all the tree operations listed in Table 1 in $O(1)$ time.*

6 Distance Oracles and Interval Graph Representations

In this section, we present new time- and space-efficient distance oracles for interval graphs and subclasses. Proofs omitted from this section can be found in Appendix D.

Interval Graphs We augment the data structure of Acan et al. [1] with the **distance** operation. We recall their main result; here (and throughout this paper), vertices of an interval graph are labeled $1, \dots, n$, sorted by the left endpoints of their intervals.

► **Lemma 7** (Succinct interval graphs, [1]). *An interval graph can be represented in $n \log n + (3 + \varepsilon)n + o(n)$ bits to support **adjacent** and **degree** in $O(1)$ time, **neighborhood** in $O(\text{degree}(v))$ time and **spath**(u, v) in $O(\text{distance}(u, v))$ time. Moreover, the interval $I_v = [l_v, r_v] \in [2n]^2$ representing a vertex can be retrieved in $O(1)$ time.*

As interval graphs are a subclass of chordal graphs, we will be using the algorithm of Munro and Wu [27] to compute distances. For a vertex v , consider the bag of v by $B_v = \{w : l_w \in I_v\}$, the set of nodes whose intervals intersect the left endpoint of v . As in [27], we define $s_v = \min B_v$. The shortest path algorithm given in [27] is similar to the one in [1]. Given $u < v$, we compute the shortest path by checking if u and v are adjacent. If so, add u to the path; otherwise, add s_v to the path and recursively find **spath**(s_v, u).

As the steps for every vertex v is the same regardless of destination u , we may store the unique step information for each vertex in a tree. We construct a tree T as follows: for every vertex $v = 1, \dots, n$, add node v to the tree as the rightmost (last) child of s_v . For example, refer to figure 1 in Appendix C. The node $v = 1$ is the root of the tree. Thus we have identified each vertex of G with a node of T . This correspondence is captured by the following important property:

► **Lemma 8.** *Let a_1, a_2, \dots, a_n be a breadth-first traversal of T . Then the corresponding vertices of G are $1, 2 \dots n$.*

With this correspondence, we will abuse notation when the context is clear and refer to both the vertex in the graph and the corresponding tree node by v . Any conversion that need to be done will be done implicitly using **node_rank**_{LEVEL} and **node_select**_{LEVEL}. Now consider the shortest path computation for $u < v$. The only candidates potentially adjacent to u are the ancestors of v at depths **depth**(u) $- 1$, **depth**(u), and **depth**(u) $+ 1$. Thus the distance algorithm is given by: for vertices $u < v$, compute the ancestor (w) of v at depth **depth**(u) $+ 1$ using **anc**. Find the distance between u and w using the **spath** algorithm. This is at most 3 steps, so is $O(1)$. Finally take the sum of the distances, one from the difference in depth and the other from the **spath** algorithm. The extra space needed is to store the tree T , using $2n + o(n)$ bits. The results described above can be summarized in the following theorem:

► **Theorem 9** (Succinct interval graphs with distance). *An interval graph G can be represented using $n \log n + (5 + \varepsilon)n + o(n)$ bits to support **adjacent**, **degree** and **distance** in $O(1)$ time, **neighborhood** in $O(\text{degree}(v))$ time and **spath**(u, v) in $O(\text{distance}(u, v))$ time.*

Finally we note that this augmentation can without changes be applied to subclasses of interval graphs. For k -proper (improper) graphs, this yields a succinct data structure whenever $k = \omega(1)$. For proper interval graphs, however, this approach does not yield a succinct data structure.

Proper Interval Graphs Recall that a proper interval graph is an interval graph where no two intervals are properly contained in each other. The information-theoretic lower bound for this class of graphs is $2n - O(\log n)$ bits [16]. Thus naively adding the distance tree to the representation uses too much space for the representation to be succinct. However, the key insight here is that the graph can be recovered from just the distance tree, and indeed, we can answer all graph queries directly on the distance tree.

Suppose that each vertex v is associated with an interval I_v and that the vertices are labeled $1, \dots, n$, sorted by their left endpoint. The neighborhood of each vertex can be succinctly described:

► **Lemma 10.** *Let v be a vertex in a proper interval graph. Then there exists vertices $u_1 \leq u_2$ such that the (closed) neighbourhood of v is the vertices in $[u_1, u_2]$.*

Let T be the tree constructed in the previous section. We already showed how to compute **spath** and **distance** for G (based on an implementation of **adjacent**). We now show how to compute **adjacent**, **degree** and **neighborhood**.

- **adjacent:** Let $u < v$. We compute s_v (using **parent**) and check whether $s_v \leq u$. If so, then u and v are adjacent. Correctness follows from the fact that the neighborhood of v is a contiguous interval.
- **neighborhood:** Let the neighborhood of v be $[u_1, u_2]$. By the definition of s_v , we have that $u_1 = s_v$. Thus it remains to compute u_2 .

► **Lemma 11.** *If v is a leaf, then $u_2 = \text{last_child}(\text{prev_internal}(v))$; otherwise we have $u_2 = \text{last_child}(v)$.*

Thus the neighborhood of v can be found in $O(1)$ time.

- **degree:** $|\text{neighborhood}(v)| = \text{degree}(v)$ can be found in $O(1)$ time by computing $u_2 - u_1$ for u_1, u_2 from **neighborhood**(v).

The results in this section are summarized in the following theorem:

► **Theorem 12** (Succinct proper interval graphs with distance). *A proper interval graph can be represented in asymptotically optimal $2n + o(n)$ bits while supporting **adjacent**, **degree**, **neighborhood** and **distance** in $O(1)$ time, and **spath**(u, v) in $O(\text{distance}(u, v))$ time.*

7 Conclusion

We present succinct data structures and distance oracles for interval graphs and several of their subclasses. All are based on the solution of a much more fundamental data-structuring problem: translating between breadth-first ranks and depth-first ranks of nodes in a tree. Apart from demonstrating the unmatched versatility of tree covering – the only method for space-efficient representations of trees known to support this BFS-DFS mapping – level-order operations are likely to find further applications in space-efficient data structures.

Regarding open questions, we note that one operation that is supported by standard tree covering has unwaveringly resisted all our attempts to be realized on top of tree slabbing: generating $\lg n$ consecutive bits of the BP or DFUDS of the tree. Such operations are highly desirable as they allow immediate reuse of any auxiliary data structures to support operations on the basis of BP resp. DFUDS. These sequences are inherently depth-first, though, and seem utterly incompatible with slicing the tree horizontally: the sought $\lg n$ bits might span a large number of (tier-2) slabs. How and if level-order rank/select and generating a word of BP or DFUDS can be simultaneously supported to run in constant time remains an open question.

A Operations Supported by Different Tree Representations

A more complete survey of the BP, DFUDS and TC representations of ordinal trees is given here, along with a table comparing the different techniques.

As the oldest tree representation after LOUDS, the BP representations have a long history and the support for many operations were added for different applications. Munro and Raman [23] first designed a BP-based representation that supports operations `parent`, `nbdesc`, `node_rankPRE/POST` and `node_selectPRE/POST` in $O(1)$ time and `child(x, i)` in $O(i)$ time. This is augmented by Munro et al. [24] to support operations related to leaves in constant time, including `leaf_rank`, `leaf_select`, `leftmost_leaf` and `rightmost_leaf`, which are used to represent suffix trees succinctly. Later, Chiang et al. [9] showed how to support `degree` using the BP representation in constant time which is needed for succinct graph representations, while Munro and Rao [25] designed $O(1)$ -time support for `anc`, `level_pred` and `level_succ` to represent functions succinctly. Constant-time support for `child`, `child_rank`, `height` and `LCA` is then provided by Lu and Yeh [21], that for `node_rankIN` and `node_selectIN` by Sadakane [33] in their work of encoding suffix trees, and that for `level_leftmost` and `level_rightmost` by Navarro and Sadakane [29].

Benoit et al. [6] were the first to represented a tree succinctly using DFUDS, and their structure supports `child`, `parent`, `degree` and `nbdesc` in constant time. This representation is augmented by Jansson et al. [20] to provide constant-time support for `child_rank`, `depth`, `anc`, `LCA`, `leaf_rank`, `leaf_select`, `leftmost_leaf` and `rightmost_leaf`, `node_rankPRE` and `node_selectPRE`. To design succinct representations of labeled trees, Barbay et al. [4] further gave $O(1)$ -time support for `node_rankDFUDS` and `node_selectDFUDS`.

TC was first used by Geary et al. [17] to represent a tree succinctly to support `child`, `child_rank`, `depth`, `anc`, `nbdesc`, `degree`, `node_rankPRE/POST` and `node_selectPRE/POST` in constant time. He et al. [18] further showed how to use TC to support all other operations provided by BP and DFUDS representations in constant time, except `node_rankIN` and `node_selectIN` which appeared after the conference version of their work. Later, based on a different tree covering algorithm, Farzan and Munro [15] destined a succinct representation that not only supports all these operations but also can compute an arbitrary word in a BP or DFUDS sequence in $O(1)$ time. The latter implies that their approach can support all the operations supported by BP or DFUDS representations.

B Further Tree Operations

In this appendix, we sketch how to support the remaining operations from Table 1. We begin with the full description of `node_rankPRE`.

node_rank_{PRE}: Given $\tau(v) = \langle \tau_1, \tau_2, \tau_3 \rangle$, find the global preorder rank. Let again u and u' be the tier-1 resp. tier-2 preorder changers associated with v . The idea is to compute the preorder rank as $u + (u' - u) + (v - u')$, i.e., the global preorder of u and the distances between u and u' resp. u' and v . Of course, we do not know u and u' or their distances directly, but we can compute them.

These distances can be stored as follows. For that we use the τ -order of nodes to store the mapping from τ -name of tier-1 preorder changers to their global preorder ranks. For each tier-2 changer, we store the mapping of τ -names to distances to associated tier-1 changers ($O(\log \log n)$ bits each).

It remains to compute $\tau(u)$ and $\tau(u')$ from $\tau(v)$. v and u' only differ in τ_3 and we use the micro-tree lookup table to store τ_3 of each node's tier-2 changer. Then, we store for each

operations	BP	DFUDS	<i>previous</i> TC	our work
child, child_rank	✓	✓	✓	✓
depth, anc, LCA	✓	✓	✓	✓
nbdesc, degree	✓	✓	✓	✓
height	✓		✓	✓
leftmost_leaf, rightmost_leaf	✓	✓	✓	✓
leaf_rank, leaf_select	✓	✓	✓	✓
level_leftmost, level_rightmost	✓		✓	✓
level_pred, level_succ	✓		✓	✓
node_rank _{PRE} , node_select _{PRE}	✓	✓	✓	✓
node_rank _{POST/IN} , node_select _{POST/IN}	✓		✓	✓
node_rank _{DFUDS} , node_select _{DFUDS}		✓	✓	✓
node_rank _{LEVEL} , node_select _{LEVEL}				✓
prev_internal, next_internal				✓

■ **Table 2** Operations supported in constant time by different succinct tree representations.

tier-2 changer u' the pair $\langle \tau_2, \tau_3 \rangle$ of its tier-1 changer (another $O(\log \log n)$ bits each). Using the τ -names of u and u' , we obtain the preorder rank of v .

In the rest of this appendix, we will sketch the tree operations that are not immediately needed for the computation of distances in interval graphs. We often show how to support an operation by describing the changes we need to make to the approach used in previous work of TC.

child, child_rank: For **child**, no changes are necessary, as we will never be getting a child of a dummy root. As for **child_rank**, the only difference occurs when we need to find the rank of an s -node. Its rank in the mini(micro)-tree is wrong because of the dummy root. For the tier-1 s -nodes, we store a bit-vector storing a 1 whenever the preceding s -node has a different parent. The **child_rank** would be distance to the preceding 1 in the bit-vector. The length of the bit-vector is the number of tier-1 s -nodes which is $O(n/H)$. Similarly for tier-2 s -nodes.

degree, nbdesc: No changes are necessary for **degree** or **nbdesc**.

height: For a mini-tree root, we may explicitly store the height. For each tier-1 s -node, we may also explicitly store the height. Now we describe how to find the height of a micro-tree root. For a micro-tree root, we store the micro-tree that contains the deepest descendant. If this micro-tree has a tier-1 promoted s -node, we store the promoted s -node with the greatest height. The height of the micro-tree root can be found by the difference in depths of the two micro-tree roots, plus the height of the tier-1 s -node. For a node that is not a micro-tree root, we consider the micro-tree μ_j^i that it is in. Suppose that μ_j^i does not contain any tier-2 promoted s -nodes. Then we proceed in the same way as in [14]. Otherwise, using the lookup table, we find the range of tier-2 promoted s -nodes that are descendants, and using a range-maximum query, find the tier-2 promoted s -node that has the greatest depth. To find the depth of a tier-2 s -node, we store the micro-tree containing the deepest descendant as in the root case. We then proceed in the same manner. The space required for range-maximum queries on all tier-2 s -nodes is linear in the number which is $O(n/H')$.

leftmost_leaf, rightmost_leaf: This is done in the same way as previously. The only difference is that we need to store the left most/right most leaf at every tier-1 *s*-node. We also need to store the micro-tree that contains the left most/right most leaf, or the micro-tree containing the relevant tier-1 *s*-node at every tier-2 *s*-node.

leaf_size: At each tier-1 *s*-node we store the number of leaves in the subtree rooted at the *s*-node. We also store the prefix sum of these values (the sum of the number of leaves from the first *s*-node to the current *s*-node). For tier-2 *s*-nodes, we store the number of leaves in the subtree of the mini-tree rooted at the *s*-node. We do not include tier-1 *s*-nodes (which are leaves of the mini-tree) in this count. For the *s*-nodes of each mini-tree, we store the prefix-sum of the number of leaves (starting from the first tier-2 *s*-node of the mini-tree to the current *s*-node).

To find the number of leaves below a node, we find the number of leaves in the micro-tree using the lookup table. We find the range of the tier-2 *s*-nodes below it, if the micro has any tier-2 promoted *s*-nodes. If not we check the unique outgoing edge if necessary for tier-2 promoted *s*-nodes. From the range of the promoted *s*-nodes, we sum of the leaves in the mini-tree from the prefix sum data structure. We also find the tier-1 *s* nodes below in similar fashion. We then take the sum of the sizes of the tier-1 *s*-nodes using the prefix-sum data structure. **leaf_rank and leaf_select:** **leaf_select** is done in the same way as before, using the compressed bit vector approach. For **leaf_rank**, in addition to the information stored, we also need to store the number of leaves preceding tier-1 *s*-nodes. For tier-2 *s*-nodes, we store the preceding tier-1 *s*-node, and the number of leaves between them.

level_leftmost, level_rightmost: No changes needed w.r.t. previous work.

level_succ, level_pred: Using **node_rank_{LEVEL}** and **node_select_{LEVEL}**, these are now trivial to implement.

LCA: The technique of He et al. [18] works for tree slabbing, too. The only change we need to make is to include tier-1 *s*-nodes in the tier-1 macro tree and tier-2 *s*-nodes in each tier-2 macro tree. These will be included instead of the dummy root added.

node_rank_{PRE/POST/IN/DFUDS/LEVEL}, node_select_{PRE/POST/IN/DFUDS/LEVEL}: For the traversals not included in the paper itself, the way to handle them is similar to how they are handled in the paper.

C Examples

This appendix shows an exemplary interval graph.

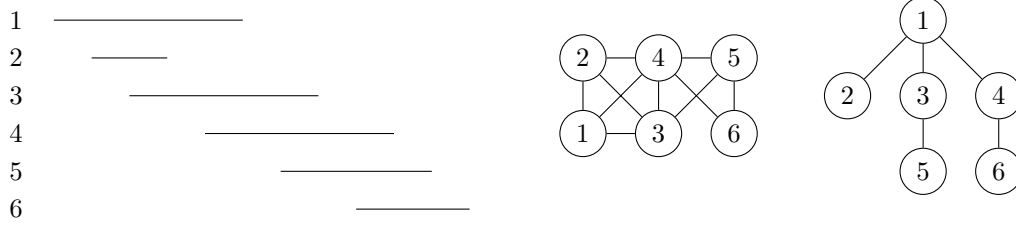


Figure 1 An Interval Graph (middle) with Interval Representation (left), and distance tree constructed (right).

D Omitted Proofs

In this appendix, we give the full proofs that are omitted from the main body due to space constraints.

Proof of Lemma 4. We focus on tier 1; tier 2 is similar. Lemma 3 already contains all ingredients: A skinny-slab subtree consists of an entire slab, so its nodes appear contiguous in level order. Each skinny mini tree thus contributes only 1 level-order changer, for a total of $O(n/H)$. For the fat subtrees, each level appears contiguously in level order, and within a level, the nodes from one mini tree form at most 3 intervals: one gap can result from a child of the root that is in another subtree, splitting the list of root children into two intervals, and a second gap can result from the single external edge. The other connections to other mini trees are through s-nodes, and hence all lie on the same level. So each fat mini tree contributes at most 3 changers per level it spans, for a total of $O(H \cdot n/B)$ level-order changers. ◀

Proof of Lemma 8. For vertices $u < v$, we will show that the node in T corresponding to u appears before the corresponding node to v in level-order.

Suppose by contradiction that it is not. Thus we must have that $s_v < s_u$ in order for it to be before u in the breadth first ordering. If $s_v = s_u$, then they are siblings and v is added to the right of u by construction.

Therefore, we have the following facts: i) $l_v > l_u$ as $v > u$, ii) $l_v \in I_{s_v}$ by definition of s_v , iii) $l_u \in I_{s_u}$ by definition of s_u , and iv) $l_{s_v} < l_{s_u}$ as $s_v < s_u$. Thus we have $l_{s_v} < l_{s_u} < l_u < l_v < r_{s_v}$, and thus $l_u \in I_{s_v}$. By definition, $s_v \in B_u$ which contradicts the fact that $s_u = \min B_u$. ◀

Proof of Lemma 10. Let $u_1 < v$ be adjacent to v . Let $w = u_1 + 1$. As G is a proper interval graph, we have the following inequalities: $l_{u_1} < l_w \leq l_v < r_{u_1} < r_w$. Thus I_v intersects I_w and v is adjacent to w . So the neighborhood of v consisting of vertices with smaller label forms a contiguous interval.

Similarly, the same argument can be made for the vertices with larger labels. ◀

Proof of Lemma 11. In the case that v is not a leaf in T , we claim that u_2 is the last child of v . Denote this child by w . Clearly v is adjacent in G to all of its children by definition.

The parent of $w + 1$ is larger than v , and thus $w + 1$ cannot be adjacent to v by the definition of T .

If v is a leaf of T , we claim that u_2 is the last child of the first internal (non-leaf) node before v in level-order. Let w denote this node. By definition, $s_w < v$ and $w \geq v$. As the neighborhood of w forms a contiguous interval, w is adjacent to v . Now consider $w + 1$. By definition of w , its level-order successor $w + 1$ must have parent $s_{w+1} > v$. Thus by the previous argument, it cannot be adjacent to v . ◀

References

- 1 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct data structures for families of interval graphs. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, pages 1–13, 2019. doi:10.1007/978-3-030-24766-9_1.
- 2 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Meeting on Algorithm Engineering & Experiments (ALENEX)*, ALENEX '10, pages 84–97. SIAM, 2010.
- 3 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 735–744. ACM, 2000. doi:10.1145/335305.335410.
- 4 Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7:52:1–52:27, September 2011. doi:http://doi.acm.org/10.1145/2000807.2000820.
- 5 Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, June 2004. doi:10.1016/j.tcs.2003.05.002.
- 6 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. doi:10.1007/s00453-004-1146-6.
- 7 Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003.
- 8 Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Succinct representations of planar maps. *Theoretical Computer Science*, 408(2-3):174–187, 2008.
- 9 Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005. doi:10.1137/S0097539702411381.
- 10 Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 118–129, 1998.
- 11 D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996. doi:10.5555/313852.314087.
- 12 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11(2):177–189, March 2017. doi:10.1007/s11786-017-0294-4.
- 13 Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In *16th Annual European Symposium on Algorithms*, pages 393–404, 2008.

- 14 Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, June 2014. doi:10.1007/s00453-012-9664-0.
- 15 Arash Farzan, Rajeev Raman, and S. Srinivasa Rao. Universal succinct representations of trees? In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming*, volume 5555 of *Lecture Notes in Computer Science*, pages 451–462, 2009.
- 16 S.R. Finch and G.C. Rota. *Mathematical Constants*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. URL: <https://books.google.ca/books?id=DL5iVYN0Ea0C>.
- 17 Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, October 2006. doi:10.1145/1198513.1198516.
- 18 Meng He, J. Ian Munro, and Srinivasa Satti Rao. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms*, 8(4):1–32, September 2012. doi:10.1145/2344422.2344432.
- 19 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.
- 20 Jesper Jansson, Kunihiro Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, March 2012. doi:10.1016/j.jcss.2011.09.002.
- 21 Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4:28:1–28:13, July 2008. doi:http://doi.acm.org/10.1145/1367064.1367068.
- 22 J. Ian Munro and Patrick K. Nicholson. Succinct posets. *Algorithmica*, 76(2):445–473, 2016.
- 23 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, January 2001. doi:10.1137/s0097539799364092.
- 24 J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001. doi:10.1006/jagm.2000.1151.
- 25 J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 1006–1015, 2004. doi:10.1007/978-3-540-27836-8_84.
- 26 J. Ian Munro and Corwin Sinnamon. Time and space efficient representations of distributive lattices. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 550–567. SIAM, 2018.
- 27 J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, pages 67:1–67:12, 2018. doi:10.4230/LIPIcs.ISAAC.2018.67.
- 28 Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.
- 29 Gonzalo Navarro and Kunihiro Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):1–39, may 2014. doi:10.1145/2601073.
- 30 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014.
- 31 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43–es, nov 2007. doi:10.1145/1290672.1290680.
- 32 R. Ravi, Madhav V. Marathe, and C. Pandu Rangan. An optimal algorithm to solve the all-pair shortest path problem on interval graphs. *Networks*, 22(1):21–35, 1992.

- 33 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.
- 34 Gaurav Singh, N. S. Narayanaswamy, and G. Ramakrishna. Approximate distance oracle in $o(n^2)$ time and $o(n)$ space for chordal graphs. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, volume 8973 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2015.
- 35 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.
- 36 Nicholas C. Wormald. Counting labelled chordal graphs. *Graphs and Combinatorics*, 1(1):193–200, 1985. doi:10.1007/BF02582944.
- 37 Peisen Zhang, Eric A. Schon, Stuart G. Fischer, Eftihia Cayanis, Janie Weiss, Susan Kistler, and Philip E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA. *Computer Applications in the Biosciences*, 10(3):309–317, 1994. doi:10.1093/bioinformatics/10.3.309.
- 38 Uri Zwick. Exact and approximate distances in graphs - A survey. In Friedhelm Meyer auf der Heide, editor, *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, volume 2161 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2001.