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

Meng He
Dalhousie University, Canada
mhe@cs.dal.ca

J. Ian Munro
University of Waterloo, Canada
imunro@uwaterloo.ca

Yakov Nekrich
Michigan Tech, USA
yakov@mtu.edu

Sebastian Wild
University of Liverpool, UK
wild@liverpool.ac.uk

Kaiyu Wu
University of Waterloo, Canada
k29wu@uwaterloo.ca

Abstract

We present the first succinct distance oracles for (unweighted) interval graphs and related classes of graphs, using a novel succinct data structure for ordinal trees that supports the mapping between preorder (i.e., depth-first) ranks and level-order (breadth-first) ranks of nodes in constant time. Our distance oracles for interval graphs also support navigation queries – testing adjacency, computing node degrees, neighborhoods, and shortest paths – all in optimal time. Our technique also yields optimal distance oracles for proper interval graphs (unit-interval graphs) and circular-arc graphs. Our tree data structure supports all operations provided by different approaches in previous work, as well as mapping to and from level-order ranks and retrieving the last (first) internal node before (after) a given node in a level-order traversal, all in constant time.

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

Keywords and phrases succinct data structures, distance oracles, ordinal tree, level order, breadth-first order, interval graphs, proper interval graphs, succinct graph representation

Related Version A shortened version appeared in ISAAC 2020, doi:10.4230/LIPIcs.ISAAC.2020.57.

Funding *J. Ian Munro*: NSERC and the Canada Research Chairs Programme

1 Introduction

As a result of the rapid growth of electronic data sets, memory requirements become a bottleneck in many applications as performance usually drops dramatically as soon as data structures do no longer fit into faster levels of the memory hierarchy in computer systems. Research on *succinct data structures* has lead to optimal-space data structures for many types of data [31].

Graphs are one the most widely used types of data. In this paper, we study *succinct distance oracles*, i.e., data structures that efficiently compute the length of a shortest path between two nodes, for interval graphs and related classes of graphs. Interval graphs are the intersection graphs of intervals on the real line and have applications in operations research [4] and bioinformatics [40]. Distance oracles are widely studied; for an overview of the extensive literature see [38, 41, 39, 34].

Our distance oracles make fundamental use of (rooted) *trees*. Standard pointer-based representations of trees use $O(n)$ words or $O(n \log n)$ bits to represent a tree on n nodes, but as the culmination of extensive work [21, 12, 26, 27, 10, 28, 24, 36, 32, 7, 22, 5, 18, 20, 16], ordinal trees can be represented *succinctly*, i.e., using the optimal $2n + o(n)$ bits of space, while supporting a plethora of navigational operations in constant time (on a word-RAM, which we assume throughout this paper); cf. Table 1. One operation that has gained some notoriety for not being supported by any of these data structures is mapping between *preorder* (i.e., depth-first) ranks and *level-order* (breadth-first) ranks of nodes. Known approaches to represent trees are either fundamentally breadth first – like the *level-order unary degree sequence* (`LOUDS`) [21] – and very limited in terms of supported operation, or they are depth first – like the *depth-first unary degree sequence* (`DFUDS`) [7], the *balanced-parentheses* (`BP`) encoding [26] and *tree covering* (`TC`) [18] – and do not support level-order ranks, (see Section 4.1 for more discussion).

In this paper, we present a new tree data structure that bridges the dichotomy, solving an open problem of [20]. Our tree data structure is based on a novel way to (recursively) decompose a tree into *forests* of subtrees that makes computing level-order information possible. We describe how to support all operations of previous `TC` data structures based on our new decomposition.

Supporting the mapping to and from level-order ranks was the missing keystone for our succinct distance oracles for interval graphs, and our tree data structure will likely be of independent interest as a building block for future work.

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:

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

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

- `node_rank`$_{\texttt{LEVEL}}(v)$ and `node_select`$_{\texttt{LEVEL}}(i)$: computing the position of node v in a level-order traversal of the tree resp. finding the ith node in the level-order traversal;
- `prev_internal`(v) and `next_internal`(v): the non-leaf node closest to v in level-order that comes before resp. after v.

Previously, `node_rank`$_{\texttt{LEVEL}}$ and `node_select`$_{\texttt{LEVEL}}$ were only supported by the `LOUDS` representation of trees [21], 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) ranks in constant time. 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*). Circular-arc graphs are intersection graphs of arcs on a circle. 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, k-proper/k-improper graphs, and circular-arc graphs in $n \lg n + (5+\varepsilon)n + o(n)$, $2n + o(n)$, $2n \lg k + 8n + o(n \log k)$, and $n \lg n + o(n \lg n)$ bits, respectively, where n is the number of vertices and $\varepsilon > 0$ is an arbitrarily small constant, such that the following operations are supported (time for interval graphs):

- `degree`(v): the degree of v, i.e., the number of vertices adjacent to v;
- `adjacent`(u, v): whether vertices u and v are adjacent;
- `neighborhood`(v): iterating through the vertices adjacent to v;
- `spath`(u, v): listing a shortest path from vertex u to v;
- `distance`(u, v): the length of the shortest path from u to v;

All query times match those of Acan et al.; `distance` has the same complexity as `adjacent`; (see Section 6 for precise statements). Succinctness of our representations (except k-(im)proper interval graphs) is evidenced by information-theoretic lower bounds of $n \lg n - 2n \lg\lg n - O(n)$ bits [17, 1] and $2n - O(\log n)$ bits [19, Thm. 12] on representing interval graphs (and circular-arc graphs) and proper interval graphs, respectively.

The best previous distance oracles for interval graphs, proper interval graphs and circular-arc graphs all result from corresponding *distance labelings*, a distributed version of distance oracles, due to Gavoille et al. [17]. They require asymptotically $\sim 5n \lg n$, $\sim 2n \lg n$, resp. $\sim 10n \lg n$ bits to represent the labeled graph. We improve all of these results even when adding $n \lg n$ bits to store node labels, and our data structures further support operations beyond `distance`. Interestingly, our distance oracles also prove *separations* between distance labelings and distance oracles: Our data structures beat corresponding lower bounds for the lengths of distance labelings – $3 \lg n - 4 \lg\lg n$ for interval graphs [17, Thm. 2] resp. $2 \lg n - 2 \lg\lg n - O(1)$ for proper interval graphs [17, Thm. 3] – showing that these "centralized" data structures are strictly more powerful than distributed ones.

2 Related Work

Succinct Representations of Ordinal Trees. The `LOUDS` representation, first proposed by Jacobson [21] and later studied by Clark and Munro [12] 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 [26, 27, 10, 28, 24, 36, 32] to include all standard operations, bar the level-order ones and $\texttt{node\_rank}_{\texttt{DFUDS}}$ / $\texttt{node\_select}_{\texttt{DFUDS}}$. The other representations have a similar history, albeit shorter, and we refer to [7, 22, 5] for DFUDS and [18, 20, 16] for TC. A full survey is also given in Appendix A; Table 2 there summarizes the operations supported by each of these three approaches.

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) [15, 13]. Other than supporting more operations, work has been done for alternative goals such as achieving compression [22, 15], reducing redundancy [32] and supporting updates [32].

Succinct Representations of Graphs. Several succinct representations of (subclasses of) graphs have been studied, e.g., for general graphs [14], k-page graphs [21], certain classes of planar graphs [11, 10, 9], separable graphs [8], posets [25] and distributive lattices [29]. Recently, Acan et al. [1] showed how to represent an *interval graph* on n vertices in $n \lg n + (3+\varepsilon)n + o(n)$ bits to support `degree` and `adjacent` in $O(1)$ time, $\texttt{neighborhood}(v)$ in $O(\texttt{degree}(v))$ time and $\texttt{spath}(u, v)$ in $O(|\texttt{spath}(u, v)|)$ time, where ε is a positive constant that can be arbitrarily small. To show the succinctness of their solution, they proved that $n \lg n - 2n \lg\lg n - O(n)$ 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 $2n \lg k + 6n + o(n \log k)$ bits, respectively, supporting the same queries.

Distance Oracles. Ravi et al. [35] considered the problem of solving the all-pair shortest path problem over interval graphs in optimal $O(n^2)$ time in 1992. Later, Gavoille and Paul in 2008 [17] designed a labeling scheme on the vertices using $5 \lg n + 3$ bit labels to compute the distance between any two vertices u, v of an interval graph in $O(1)$ time. Their work implies a $5n \lg n + O(n)$ bit distance oracle by simply concatenating all labels. Furthermore, they proved a $3 \lg n - o(\lg n)$ bit lower bound for distance labeling. On the subject of chordal graphs (which contain interval graphs), Singh et al. [37] designed a data structure of $O(n)$ words that can *approximate* the distance between two vertices u and v in $O(1)$ time, and the answer is between $|\texttt{distance}(u, v)|$ and $2|\texttt{distance}(u, v)| + 8$. More recently, Munro and Wu [30] designed a succinct representation of chordal graphs using $n^2/4 + o(n^2)$ bits, which inspired our new distance oracles. They also designed an *approximate* distance oracle of $n \lg n + o(n \log n)$ bits with $O(1)$ query time, where answers are within 1 of the actual distance.

3 Notation and Preliminaries

We write $[n..m] = \{n, \ldots, m\}$ 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 Pătraşcu [33] for compressed bitvectors:

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

- $\mathtt{access}(\mathcal{B},i)$: *return $\mathcal{B}[i]$, the bit at index i in $\mathcal{B}$.*
- $\mathtt{rank}_\alpha(\mathcal{B},i)$: *return the number of bits with value $\alpha \in \{0,1\}$ in $\mathcal{B}[1..i]$.*
- $\mathtt{select}_\alpha(\mathcal{B},i)$: *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 $1,\ldots,n$ (order of appearance) in a preorder traversal. Tree covering (TC) relies on a two-tier decomposition: the tree consists of mini trees, each of which consists of micro trees. The former will be denoted by μ^i, the latter by μ^i_j.

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 [18]. However, more carefully designed procedures yield restrictions on the touching points of subtrees:

▶ **Lemma 2** (Tree Covering, [15, Thm. 1]). *For any parameter $B \ge 3$, an ordinal tree with n nodes can be decomposed, in linear time, into connected subtrees with the following properties.*

(a) *Subtrees are pairwise disjoint except for (potentially) sharing a common subtree root.*
(b) *Each subtree contain at most $2B$ nodes.*
(c) *The overall number of subtrees is $\Theta(n/B)$.*
(d) *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.*

Inspecting 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: a node whose subtree size is greater than B, 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, where w is 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 $\mathtt{height}(\mu^i)\cdot\lg n$ bits for $\mathtt{height}(\mu^i)$ 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 $|\mu^i| = \omega(\mathtt{height}(\mu^i)\log n)$ nodes or height $\mathtt{height}(\mu^i) = o(|\mu^i|/\log n)$. Since the tree T to be stored 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 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

We fix two parameters: $H \in \mathbb{N}$, the $\underline{h}$eight of slabs, and $B > H$, the target $\underline{b}$lock size. We start by cutting T horizontally into slabs of thickness/height exactly H, but we allow ourselves to start cutting at an offset $o \in [H]$. 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* ("$\underline{s}$labbed 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 $S_i = \big\{v : \texttt{depth}(v) \in [(i-1)H + o \ \ .. \ \ iH + o]\big\}$, the set of nodes making up the ith slab, also denotes the ith slab itself, $i = 0, \ldots, h$. Obviously, the number of slabs is $h + 1 \le n/H + 2$. 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. Note that the first slab does not contain any s-nodes and the last slab does not contain promoted s-nodes.

Since 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 $|S_i| \le B$, S_i is a *skinny* subtree (after adding the dummy root) and will not be further subdivided. 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; an example is shown in Figure 1.

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

(a) *Subtrees are pairwise disjoint except for (potentially) sharing a common subtree root.*

(b) *Subtrees have size $\le M = 2B$ and height $\le H$.*

(c) *Every subtree is either* pure *(a connected induced subgraph of T), or* glued *(a dummy root, whose children are connected induced subgraphs of T).*

(d) *Every subtree is either a* skinny (slab) *subtree (an entire slab) or* fat*.*

(e) *The overall number of subtrees is $O(n/H)$, among which $O(n/B)$ are fat.*

(f) *Connections between subtrees μ and μ' are of the following types:*

1. *μ and μ' share a common root. Each subtree contains at most two blocks of consecutive children of a shared root.*
2. *The root of μ' is a child of the root of μ.*
3. *The root of μ' is a child of another node in μ. This happens at most once in μ.*
4. *μ' contains the original copy of a promoted s-node in μ. The total number of these connections is $O(n/H)$.*

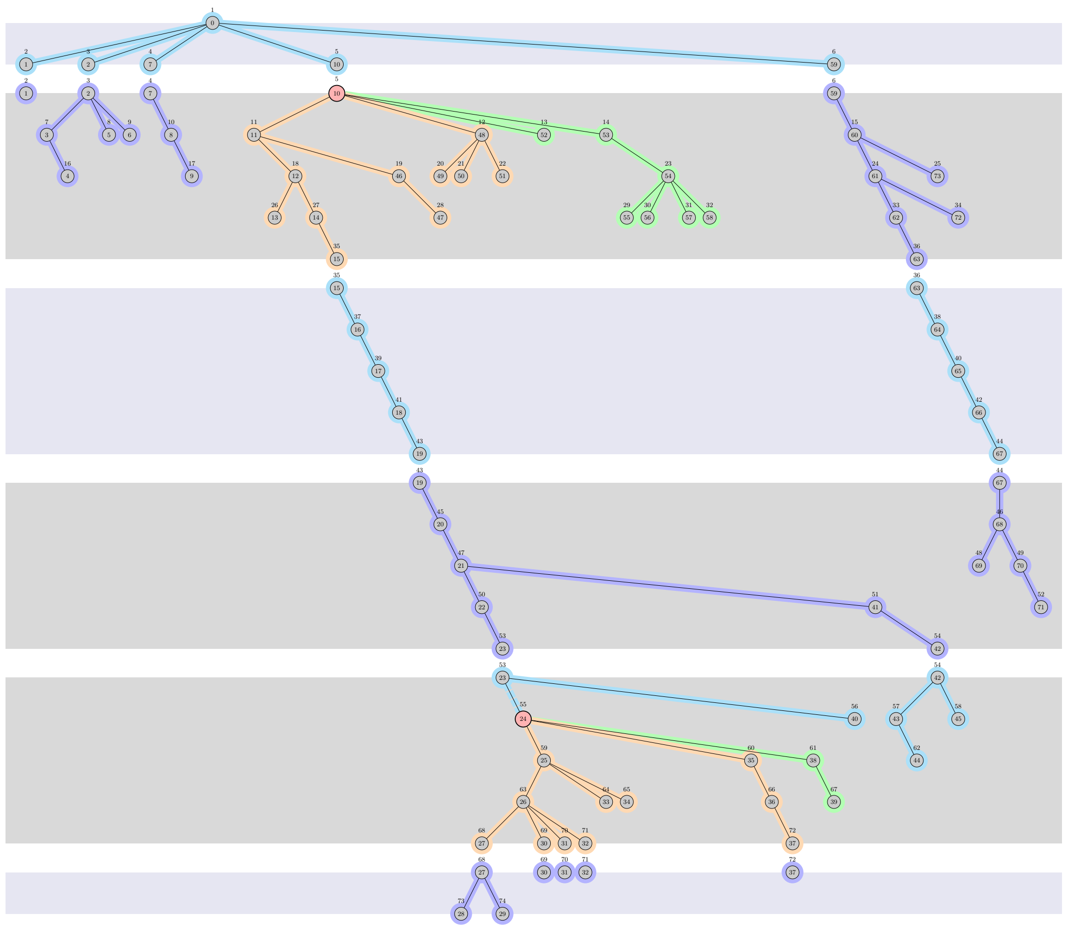


Figure 1 An example of the tree-slabbing decomposition from Theorem 3 with $B = 11$ and $H = 4$. Slabs are shown as shaded areas (light blue for skinny slabs, light gray for fat slabs). All s-nodes are depicted twice, one in each slab they belong to. The trees within a slab are connected by a dummy root (not depicted) and further decomposed as in Lemma 2; the resulting subtrees are shown by the edge colors.

„Oans, zwoa, G'suffa.“ The above tree-slabbing scheme has two parameters, H and B. 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 $\mu^1, \ldots, \mu^m$ 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 $\mu^i_1, \ldots, \mu^i_{m'_i}$ of size at most $M' = 2b = \frac{1}{4} \lg n$. The total number of micro trees is $m' = m'_1 + \cdots + m'_m = O(n/H')$, but at most $O(n/B')$ are fat micro trees. We refer to the s-nodes created at mini resp. micro tree level as *tier-1* resp. *tier-2* s-nodes. 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.

Internal node ids. Internally to our data structure, we will identify a node v by its “τ-name”, a triple specifying the mini tree, the micro tree within the mini tree, and the node within the micro tree. More specifically, $\tau(v) = \langle \tau_1, \tau_2, \tau_3 \rangle$ means that v is the τ_3th node in the micro-tree-local preorder (DFS order) traversal of $\mu_{\tau_2}^{\tau_1}$; mini trees are ordered by when their first node appears in a preorder traversal of T, ties (among subtrees sharing roots) broken by the second node, and similarly for micro trees inside one mini tree.

Since there are $O(n/H)$ mini trees, $O(B/H')$ micro trees inside one mini tree, and $O(B')$ nodes in one micro tree, we can encode any τ-name with $\sim \lg n + 2\lg\lg n + 2\lg\lg\lg n$ bits. 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* of nodes.

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*.

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 on Slabbed Trees

We now describe how to support operations efficiently in our data structure. We describe some exemplary ones here and defer the others to Appendix B.

We start by 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 $2n + O(n/H') = 2n + o(n)$ bits of space; the other components of the type are asymptotically negligible. A 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 $O(\sqrt{n}\,\mathrm{polylog}(n))$ 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* [20, 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 Theorem 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. We can afford to do this using Lemma 1 for tier-1 and tier-2 in $o(n)$ bits. (The universe grows to $n\operatorname{polylog}(n)$, but with sufficiently large c that does not affect the space by more than a constant factor). We can store $O(\log n)$ bits for each tier-1 changer and $O(\log\log n)$ bits for each tier-2 changer in an array, and using rank on the above bitvectors, we can access that information given the node's 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 node's 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 of $\tau_3(u')$ and the distance from the last 1 in the bit vector indicating tier-2 changers.

Rank 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 store them as follows. 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 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.

5.2 Level-order rank/select

Let $w_1, \ldots, w_n$ be the nodes of T in level order, i.e., w_i is the ith 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.

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

Proof. 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. ◀

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

Select Given the level-order rank i, find $\tau(w_i)$. We store $\tau_1(w_1), \ldots, \tau_1(w_n)$ in a piece-wise constant array, using the same technique as for preorder (compressed bitvector for changers, explicit values at changers), and similarly for $\tau_2(w_1), \ldots, \tau_2(w_n)$. 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 j' at all tier-2 level-order changers and compute the distance j'' of w_i from its tier-2 changer. The sum $j' + j''$ 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 compute i as $j + (j' - j) + (i - j')$ for w_j and $w_{j'}$ the tier-1 resp. tier-2 level-order changers of $v = w_i$; (this is similar as for preorder rank above).

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

5.3 Previous Internal Node 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 $\mu^{\tau_1(v)}_{\tau_2(v)}$, 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.

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

▶ **Theorem 5** (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 related classes. Here (and throughout this paper), we assume an interval realization of the graph $G = ([n], E)$ is given where all endpoints are disjoint and lie in $[2n]$; such can be computed efficiently from G [1]. Vertices of an interval graph are labeled $1, \ldots, n$, sorted by the left endpoints of their intervals.

6.1 Distances in Interval Graphs

We first describe how to augment an interval-graph representation with $O(n)$ additional bits of space to support distance in constant time. Our distance oracles are based on the graph data structures of Acan et al. [1]; we recall their result for interval graphs.

▶ **Lemma 6** (Succinct interval graphs, [1]). *An interval graph can be represented using $n \lg n + (3 + \varepsilon)n + o(n)$ bits to support* adjacent *and* degree *in $O(1)$ time,* neighborhood *in* $O(\texttt{degree}(v))$ *time and* $\texttt{spath}(u, v)$ *in* $O(\texttt{distance}(u, v))$ *time. Moreover, the interval* $I_v = [\ell_v, r_v] \in [2n]^2$ *representing a vertex can be retrieved in* $O(1)$ *time.*[1]

As interval graphs are a subclass of chordal graphs, we will be using the algorithm of Munro and Wu [30] to compute distances. For a vertex v, denote the *bag* of v by $B_v = \{w : \ell_v \in I_w\}$, i.e., the set of vertices whose intervals contain the left endpoint of v's interval. As in [30], we define $s_v = \min B_v$. The shortest path algorithm given in [30] 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 $\texttt{spath}(s_v, u)$.

As the next step for every vertex v is the same regardless of destination u, we can store this unique step for each vertex as the parent pointer of a tree. We construct a tree T as follows: for every vertex $v = 1, \ldots, n$ (in that order), add node v to the tree as the rightmost (last) child of s_v; see Figure 2 for an example. 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 Lemma 7 below.

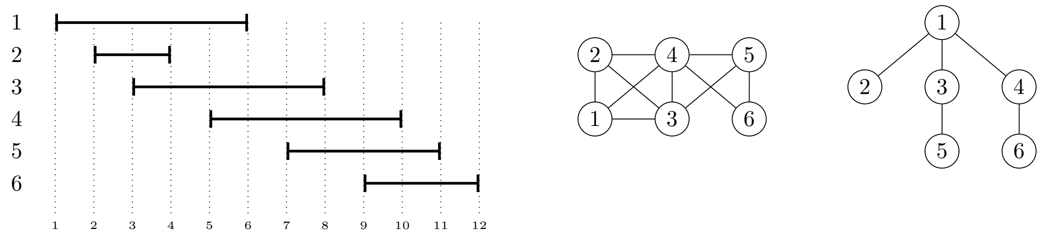


Figure 2 An Interval Graph (middle) with Interval Representation (left), and distance tree constructed (right).

We note that the above construction is undefined for a disconnected graph, as the leftmost interval of a component would have an undefined parent. The simplest way to solve this is to set the parent of such a vertex v as $v - 1$ (that is we add the edge between them). We will also need to include a length n bit-vector, where the ith entry is a 1 if vertex i is the first vertex of a component (to keep track of the edges we added). Any distance queries (between u and v) will first check if the two vertices are in the same component by performing a rank query on the bit-vector at indices u and v, and check that they are the same. Similarly for adjacency and neighborhood queries; we will need to check if vertices are the first vertex of a component, and if so, make sure the added edge is not reported.

▶ **Lemma 7** (Distance tree BFS). *Let* $a_1, a_2, \ldots, a_n$ *be a* breadth-first *traversal of* T*. Then the corresponding vertices of* G *are* $1, 2 \ldots n$*.*

Proof. First note that it immediately follows from the incremental construction of T in level order that the node with largest index inserted so far is always the rightmost node on the deepest level of T. So if the graph is disconnected, our procedure above does not change the order of the vertices in level order, nor the order of the vertices in G. So we may assume that the graph is connected.

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

[1] Note that the arXiv version [2] of [1] erroneously claims a space usage of $n \lg n + (2+\varepsilon)n + o(n)$ bits for their data structure. Interestingly, it is indeed possible to reduce the space to that by storing $r_1, \ldots, r_n \in [2n]$, the right endpoints, in rank-reduced form, $R[1..n]$, (a permutation) and using $r_i = \texttt{select}_1(S, R[i])$.

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) $\ell_v > \ell_u$ as $v > u$, ii) $\ell_v \in I_{s_v}$ by definition of s_v, iii) $\ell_u \in I_{s_u}$ by definition of s_u, and iv) $\ell_{s_v} < \ell_{s_u}$ as $s_v < s_u$. Thus we have $\ell_{s_v} < \ell_{s_u} < \ell_u < \ell_v < r_{s_v}$, and thus $\ell_u \in I_{s_v}$. By definition, $s_v \in B_u$ which contradicts the fact that $s_u = \min B_u$. ◀

With this correspondence, we will abuse notation when the context is clear and refer to both the vertex in the graph and the corresponding node in the tree by v. Any conversion that needs to be done will be done implicitly using $\texttt{node\_rank}_{\texttt{LEVEL}}$ and $\texttt{node\_select}_{\texttt{LEVEL}}$. Now consider the shortest path computation for $u < v$. The only candidates potentially adjacent to u are the ancestors of v at depths $\texttt{depth}(u) - 1$, $\texttt{depth}(u)$, and $\texttt{depth}(u) + 1$. The ancestor z of v at depth $\texttt{depth}(u) + 2$ cannot be adjacent to u as $w = \texttt{parent}(z) > u$, and $\texttt{parent}(z)$ is defined as the smallest node adjacent to z. Thus the distance algorithm reduces to the following: For vertices $u < v$, compute $w = \texttt{anc}(v, \texttt{depth}(u) + 1)$, the ancestor of v at depth $\texttt{depth}(u) + 1$. Find the distance between u and w using the `spath` algorithm. This is at most 3 steps, so in $O(1)$ time. 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, and for disconnected graphs, the component bitvector.

The results described above are summarized in the following theorem:

▶ **Theorem 8** (Succinct interval graphs with distance). *An interval graph G can be represented using $n \lg n + (5 + \varepsilon)n + o(n)$ bits to support* `adjacent`, `degree` *and* `distance` *in $O(1)$ time,* `neighborhood` *in $O(\texttt{degree}(v) + 1)$ time, and $\texttt{spath}(u, v)$ in $O(\texttt{distance}(u, v) + 1)$ time. If G is disconnected, the space needed is $n \lg n + (6 + \varepsilon)n + o(n)$ bits.*

Finally we note that this augmentation can without changes be applied to subclasses of interval graphs; we thus obtain the following theorem:

▶ **Theorem 9** (Succinct k-proper/-improper interval graphs with distance). *A k-proper (k-improper) interval graph[2] G can be represented using $2n \lg k + 8n + o(n \log k)$ bits to support* `degree`, `adjacent`, `distance` *in $O(\log \log k)$ time,* `neighborhood` *in $O(\log \log k \cdot (\texttt{degree}(v) + 1))$ time and $\texttt{spath}(u, v)$ in $O(\log \log k \cdot (\texttt{distance}(u, v) + 1))$ time. If G is disconnected, the space needed is $2n \lg k + 9n + o(n \log k)$ bits.*

The additional space is a lower-order term if $k = \omega(1)$. While Acan et al.'s data structure is not succinct, either, for $k = O(1)$, a different tailored representation for proper interval graphs ($k = 0$) is presented there. Here, simply adding our distance tree is not good enough.

6.2 Succinct Proper Interval Graphs with Distance

Recall that a proper interval graph is an interval graph that admits an interval representation with no interval properly contained in another. As before, each vertex v is associated with an interval I_v and vertices sorted by left endpoints. The information-theoretic lower bound for this class of graphs is $2n - O(\log n)$ bits [19, Thm. 12]. Hanlon also shows that asymptotically,

[2] We note that Klavík et al. [23] consider a closely related class of interval graphs, k-NestedINT that is similar to (and contains) Acan et al.'s [1] class of $(k-1)$-improper interval graphs, but defines k as the length of longest chain of pairwise nested intervals. The data structures of Acan et al. directly apply to this notion by adapting the definition of S'.

a 0.626578-fraction of all proper interval graphs is connected, so the same lower bound holds for connected proper interval graphs.

While adding the distance tree on top of the existing representation is too costly, our the key insight here is that the graph can be *recovered* from the distance tree, and indeed, we can answer all graph queries directly on the latter. Thus for connected proper interval graphs, the representation is succinct, but an extra $n + o(n)$ bits is required for disconnected proper interval graphs in the worst case. However, if the number of components is not too large, say $O(n/\log(n))$ components, our redundancy remains $o(n)$ using Lemma 1. We will assume that the graph is connected, and use the extra steps required as described in the general interval graph case. First, the neighborhood of a vertex can be succinctly described:

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

Proof. 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: $\ell_{u_1} < \ell_w \le \ell_v < r_{u_1} < r_w$. 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. ◀

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 first check if v is the leftmost node in its component; if so, u and v cannot be adjacent. Otherwise, we compute s_v (using `parent`); then u and v are adjacent iff $s_v \le u$. 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 $u_1 = s_v$ (unless v is leftmost; then $u_1 = v$). Thus it remains to compute u_2. If v is rightmost in its component, $u_2 = v$; otherwise we find u_2 using the following lemma in $O(1)$ time.

▶ **Lemma 11.** *If v is a leaf, then $u_2 =$ `last_child`(`prev_internal`(v)); otherwise we have $u_2 =$ `last_child`(v).*

Proof. 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 $w =$ `last_child`(`prev_internal`(v)) denote this node. By definition, $s_w < v$ and $w \ge v$. 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. ◀

`degree`: $|$`neighborhood`$(v)| =$ `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; we note that the succinct representation of neighbors allows to report those faster than is possible using Acan et al's representation.

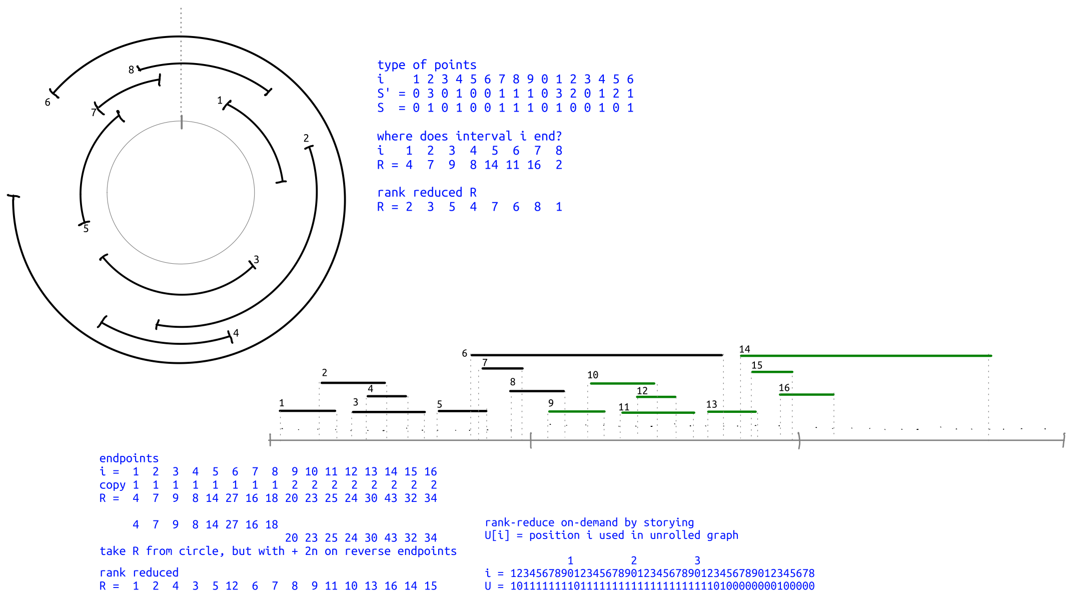
▶ **Theorem 12** (Succinct proper interval graphs with distance)**.** *A connected proper interval graph can be represented in asymptotically optimal $2n + o(n)$ bits while supporting* `adjacent`*,* `degree`*,* `neighborhood` *and* `distance` *in $O(1)$ time, and* `spath`(u, v) *in $O($*`distance`*$(u, v))$ time. A disconnected proper interval graph will use $3n + o(n)$ bits in the worst case; if the number of components is $O(n/\log n)$, then the space is still $2n + o(n)$.*

6.3 Distances in Circular-Arc Graphs

We finally show how to extend our distance oracles to circular-arc graphs. We follow the notation of [1] for circular-arc graphs, in particular, we assume that we are given left and right endpoints of the vertices' arcs in $[\ell_v, r_v] \in [2n]$ for $v = 1, \ldots, n$, all endpoints are distinct, and $\ell_1 < \cdots < \ell_n$, i.e., vertex ids are by sorted left endpoints. Moreover, v is a *normal* vertex if $\ell_v < r_v$; otherwise it is a *reversed* vertex corresponding to the arc $[\ell_v, 2n] \cup [1, r_v]$. We assume that G is connected; if not, G is actually an interval graph, and we can use Theorem 8.

Acan et al. [1, 2] describe two succinct data structures for circular-arc graphs: one based on succinct point grids (the "grid version") that supports all operations of Lemma 6, but each with a $\Theta(\log n/\log\log n)$-factor overhead in running time (see [1, Thm. 5] resp. [2, Thm. 6]), and a second (the "grid-less version") that does not support `degree` (other than by iterating over `neighborhood`), but handles all other queries in optimal time (see [2, Thm. 7]). We describe how to augment either of these to also answer `distance` queries (in $O(\log n/\log\log n)$ resp. $O(1)$ time) using $O(n)$ additional bits of space.

The idea of our distance oracle is to simulate access to the interval graph obtained by "unrolling" G *twice*, and then use the distance algorithm for interval graphs therein. Figure 3 shows an example.



```
type of points
i    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
S' = 0 3 0 1 0 0 1 1 1 0 3 2 0 1 2 1
S  = 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0 1

where does interval i end?
i   1  2  3  4  5  6  7  8
R = 4  7  9  8 14 11 16  2

rank reduced R
R = 2  3  5  4  7  6  8  1
```

```
endpoints
i =  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
copy 1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
R =  4  7  9  8 14 27 16 18 20 23 25 24 30 43 32 34

     4  7  9  8 14 27 16 18
                             20 23 25 24 30 43 32 34
take R from circle, but with + 2n on reverse endpoints

rank reduced
R =  1  2  4  3  5 12  6  7  8  9 11 10 13 16 14 15
```

```
rank-reduce on-demand by storying
U[i] = position i used in unrolled graph

             1         2         3
i = 123456789012345678901234567890123456789012345678
U = 101111111101111111111111111111110100000000100000
```

Figure 3 An examplary circular-arc graph and its twice-unrolled interval graph. The figure also shows some of the sequences used in Acan et al.'s succinct representations.

Gavoille and Paul [17] have shown that this construction preserves distances in the following sense:

▶ **Lemma 13** ([17, Lem. 6]). *Let $G = ([n], E)$ be a circular-arc graph with arcs $[\ell_v, r_v]$ where endpoints are distinct and in $[2n]$ and $\ell_1 < \cdots < \ell_n$. Define $\tilde{G} = ([2n], \tilde{E})$ as the interval graph with the following sets of intervals: for every normal vertex v, include $[\ell_v, r_v]$ and $[\ell_v + 2n, r_v + 2n]$ and for every reversed vertex u, include $[r_u, \ell_u + 2n]$ and $[r_u + 2n, \ell_u + 4n]$. Then for any $u < v$, we have (identifying vertices with the ranks of their left endpoints)*

$$\texttt{distance}_G(u, v) \quad = \quad \min\bigl\{\texttt{distance}_{\tilde{G}}(u, v),\ \texttt{distance}_{\tilde{G}}(v, u + n)\bigr\}.$$

Both data structures of Acan et al. store the sequences r' and r'' of the rank-reduced right endpoints for normal resp. reversed vertices, in the order of their left endpoints. Using rank/select on the bitvectors S and S' – storing the "type" of endpoints (left vs. right for S; left normal, right normal, left reversed, right reversed for S') – we can compute the endpoints $(l_v, r_v) \in [2n]^2$ of any vertex v in the same complexity as reading entries of r' and r'', i.e., $O(\log n/\log\log n)$ time for the grid version and $O(1)$ time for the grid-free version.

Given access to r, the sequence of right endpoints of the circular arcs, we can simulate access to a right endpoint $\tilde{r}_v$, $v \in [2n]$, in the twice-unrolled interval graph $\tilde{G}$ as follows: If $v \le n$ and a normal vertex, $\tilde{r}_v = r_v$. If $v \le n$ and a reversed vertex, $\tilde{r}_v = r_v + 2n$. Otherwise, $v \in [n + 1, 2n]$; then $\tilde{r}_v = \tilde{r}_{v-n} + 2n$. (See $\texttt{R}$ in Figure 3.) By storing the bitvector $U[1..6n]$ with rank support where $U[i] = 1$ iff $\tilde{\ell}_v = i$ or $\tilde{r}_v = i$ for some v, we can compute the rank-reduced intervals $[\tilde{\ell}'_v, \tilde{r}'_v]$ for all vertices $v = 1, \ldots, 2n$ of $\tilde{G}$. We also store the distance tree for $\tilde{G}$ using the data structure of Theorem 5 in $4n + o(n)$ bits, as well as the auxiliary data structures of Acan et al. (without r) from Lemma 6, all of which occupy $O(n)$ bits. Together this shows the following result.

▶ **Theorem 14.** *A circular-arc graph on n vertices can be represented in $n \lg n + o(n \lg n)$ bits of space to support either*

(a) `adjacent`, `degree`, *and* `distance` *in $O(\log n/\log\log n)$ time,*
`neighborhood`(v) *in $O((\texttt{degree}(v) + 1) \cdot \log n/\log\log n)$, and*
`spath`(u, v) *in $O((\texttt{distance}(u, v) + 1) \cdot \log n/\log\log n)$ time; or*

(b) `adjacent` *and* `distance` *in $O(1)$ time,*
`neighborhood`(v) *and* `degree`(v) *in $O(\texttt{degree}(v) + 1)$, and*
`spath`(u, v) *in $O(\texttt{distance}(u, v) + 1)$ time.*

7 Conclusion

We present succinct data structures and distance oracles for interval graphs and several related families of graphs. All are based on the solution of a fundamental data-structuring problem on trees: translating between breadth-first ranks and depth-first ranks of nodes in an ordinal 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 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 Survey of Succinct Tree Representations

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

The *level-order unary degree sequence* (`LOUDS`) representation of an ordinal tree [21] 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 [3], but a major disadvantage of `LOUDS`-based data structures is that they support only a very limited set of operations [31].

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 [7]. Other approaches that allow to support largely the same set of operations are based on the *balanced-parentheses* (`BP`) encoding [26] or rely on *tree covering* (`TC`) [18] for a hierarchical tree decomposition.

As the oldest tree representation after `LOUDS`, the `BP`-based representations have a long history and the support for many operations was added for different applications. Munro and Raman [26] first designed a `BP`-based representation supporting **`parent`**, **`nbdesc`**, $\texttt{node\_rank}_{\texttt{PRE/POST}}$ and $\texttt{node\_select}_{\texttt{PRE/POST}}$ in $O(1)$ time and $\texttt{child}(x,i)$ in $O(i)$ time. This is augmented by Munro et al. [27] 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. [10] showed how to support **`degree`** using the `BP` representation in constant time which is needed for succinct graph representations, while Munro and Rao [28] 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 [24], that for $\texttt{node\_rank}_{\texttt{IN}}$ and $\texttt{node\_select}_{\texttt{IN}}$ by Sadakane [36] in their work of encoding suffix trees, and that for **`level_leftmost`** and **`level_rightmost`** by Navarro and Sadakane [32].

Benoit et al. [7] 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. [22] to provide constant-time support for **`child_rank`**, **`depth`**, **`anc`**, **`LCA`**, **`leaf_rank`**, **`leaf_select`**, **`leftmost_leaf`** and **`rightmost_leaf`**, $\texttt{node\_rank}_{\texttt{PRE}}$ and $\texttt{node\_select}_{\texttt{PRE}}$. To design succinct representations of labeled trees, Barbay et al. [5] further gave $O(1)$-time support for $\texttt{node\_rank}_{\texttt{DFUDS}}$ and $\texttt{node\_select}_{\texttt{DFUDS}}$.

`TC` was first used by Geary et al. [18] to represent a tree succinctly to support **`child`**, **`child_rank`**, **`depth`**, **`anc`**, **`nbdesc`**, **`degree`**, $\texttt{node\_rank}_{\texttt{PRE/POST}}$ and $\texttt{node\_select}_{\texttt{PRE/POST}}$ in constant time. He et al. [20] further showed how to use `TC` to support all other operations provided by `BP` and `DFUDS` representations in constant time, except $\texttt{node\_rank}_{\texttt{IN}}$ and $\texttt{node\_select}_{\texttt{IN}}$ which appeared after the conference version of their work. Later, based on a different tree covering algorithm, Farzan and Munro [16] 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.

operations	BP	DFUDS	*previous* 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`	✓		✓	✓
$\texttt{node\_rank}_{\texttt{PRE}}$, $\texttt{node\_select}_{\texttt{PRE}}$	✓	✓	✓	✓
$\texttt{node\_rank}_{\texttt{POST/IN}}$, $\texttt{node\_select}_{\texttt{POST/IN}}$	✓		✓	✓
$\texttt{node\_rank}_{\texttt{DFUDS}}$, $\texttt{node\_select}_{\texttt{DFUDS}}$		✓	✓	✓
$\texttt{node\_rank}_{\texttt{LEVEL}}$, $\texttt{node\_select}_{\texttt{LEVEL}}$				✓
`prev_internal`, `next_internal`				✓

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

B Tree Operations

In this appendix, we sketch how to support the remaining operations from Table 1. Many techniques are similar to previous work on `TC` data structures [15, 20, 18], but most operations require some changes to work on top of tree slabbing. Operations required for our distance oracles are presented in full details to be self-contained; for the others and where appropriate, we only describe the changes necessary to the algorithms given in [15].

`parent`: $\texttt{parent}(v) = \texttt{anc}(v, 1)$, so it is subsumed by the level-ancestor solution below.

`last_child`: Obviously, this can be obtained as $\texttt{last\_child}(v) = \texttt{child}(v, \texttt{degree}(v))$ using the operations below, but it can also easily be implemented directly as follows.

Given $\tau(v)$, find $\tau(u)$, for u the rightmost child of v. Suppose first that $\tau_3(v) \neq 1$. Then all children of v are inside $\mu^{\tau_1(v)}_{\tau_2(v)}$. We use the micro-tree lookup table to obtain $\tau_3(u)$, and whether u is a promoted node. If not, we return $\langle \tau_1(v), \tau_2(v), \tau_3(u) \rangle$. For promoted nodes, we store their canonical τ-name. We store the canonical $\langle \tau_2(u), \tau_3(u) \rangle$ if u is in $\mu^{\tau_1(v)}$, and the full $\tau(u)$ otherwise, plus 1 extra bit to distinguish these cases. (This amounts to $o(n)$ extra bits as there are $O(n/H')$ tier-2 promoted nodes and $O(n/H)$ tier-1 promoted nodes.)

If $\tau_1(v) = 1 \neq \tau_2(v)$, we store $\langle \tau_2(u), \tau_3(u) \rangle$ of v' rightmost child, which must lie in $\mu^{\tau_1(v)}$. If $\tau_1(v) = 1 = \tau_2(v)$, we simply store $\tau(u)$ directly.

`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.

`anc`: Given $\tau(v)$, find $\texttt{anc}(v, i) = \tau(w)$ for w the ancestor of v on level $\texttt{depth}(v) - i$. The solution of [18, §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 15** (Level ancestors, [6, Thm. 13]). *There is a data structure using $O(n \log n)$ bits of space that answers* $\texttt{anc}(v, i)$ *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 $1, 2, \ldots, \sqrt{H}$ or $\sqrt{H}, 2\sqrt{H}, 3\sqrt{H}, \ldots, H$; 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 [18, 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 $O(1)$.

* * *

The remaining tree operations are not immediately needed for the computation of distances in interval graphs. We sketch how to support the operations by describing the changes needed 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 [15]. 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 $\texttt{node\_rank}_{\texttt{LEVEL}}$ and $\texttt{node\_select}_{\texttt{LEVEL}}$, these operations are now straight-forward and do not need a tailored implementation.

`LCA`: The technique of He et al. [20] 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.

$\texttt{node\_rank}_{\texttt{PRE/POST/IN/DFUDS/LEVEL}}$, $\texttt{node\_select}_{\texttt{PRE/POST/INDFUDS/LEVEL}}$: For other traversals can be handled similarly to preorder / level order.

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 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct data structures for families of interval graphs, 2019. arXiv:1902.09228.

3 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Meeting on Algorithm Engineering & Expermiments (ALENEX)*, ALENEX '10, pages 84–97. SIAM, 2010.

4 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.

5 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.

6 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.

7 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.

8 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.

9 Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Succinct representations of planar maps. *Theoretical Computer Science*, 408(2-3):174–187, 2008.

10 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.

11 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.

12 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.

13 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.

14 Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In *16th Annual European Symposium on Algorithms*, pages 393–404, 2008.

15 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.

16 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.

17 Cyril Gavoille and Christophe Paul. Optimal distance labeling for interval graphs and related graph families. *SIAM Journal on Discrete Mathematics*, 22(3):1239–1258, January 2008. doi:10.1137/050635006.

18 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.

19 Phil Hanlon. Counting interval graphs. *Transactions of the American Mathematical Society*, 272(2):383–383, February 1982. doi:10.1090/s0002-9947-1982-0662044-8.

20 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.

21 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.

22 Jesper Jansson, Kunihiko 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.

23 Pavel Klavík, Yota Otachi, and Jiří Šejnoha. On the classes of interval graphs of limited nesting and count of lengths. *Algorithmica*, 81(4):1490–1511, April 2019. doi:10.1007/s00453-018-0481-y.

24 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.

25 J. Ian Munro and Patrick K. Nicholson. Succinct posets. *Algorithmica*, 76(2):445–473, 2016. doi:10.1007/s00453-015-0047-1.

26 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.

27 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.

28 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.

29 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. doi:10.1137/1.9781611975031.36.

30 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.

31 Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

32 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):1–39, may 2014. doi:10.1145/2601073.

33 Mihai Patrascu. Succincter. In *Symposium on Foundations of Computer Science (FOCS)*. IEEE, October 2008. doi:10.1109/focs.2008.83.

34 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.

35 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. doi:10.1002/net.3230220103.

36 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

37 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. doi:10.1007/978-3-319-15612-5_9.

38 Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, April 2014. doi:10.1145/2530531.

39 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.

40 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.

41 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. doi:10.1007/3-540-44676-1_3.