Succinct and Compact Data Structures for Intersection Graphs

by

Kaiyu Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Gerth Stølting Brodal Professor Dept. of Computer Science, Aarhus University
Supervisor(s):	J. Ian Munro Professor Cheriton School of Computer Science, University of Waterloo
Internal Member:	Lap Chi Lau Professor Cheriton School of Computer Science, University of Waterloo
Internal Member:	Richard Peng Associate Professor Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Stephen Melczer Assistant Professor Dept. of Combinatorics and Optimization, University of Waterloo

Author's Declaration

Statement of Contributions

This thesis is based on the following publications or unpublished papers.

1) [43] Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020
In Chapter 3 and Chapter 4.

2) [26] Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, Anurag Murty Naredla, and Kaiyu Wu. Shortest beer path queries in interval graphs. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, pages 59:1–59:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022
In Chapter 5.

3) [45] Meng He, J. Ian Munro, and Kaiyu Wu. Succinct intersection graphs revisited
In Chapter 7 and Chapter 4.

4) [22] Jingbang Chen, Meng He, J. Ian Munro, Richard Peng, Kaiyu Wu, and Daniel Zhang. Distance queries over dynamic interval graphs
In Chapter 6.

Abstract

This thesis designs space efficient data structures for several classes of intersection graphs, including interval graphs, path graphs and chordal graphs. Our goal is to support navigational operations such as `adjacent` and `neighborhood` and distance operations such as `distance` efficiently while occupying optimal space, or a constant factor of the optimal space.

Using our techniques, we first resolve an open problem with regards to succinctly representing ordinal trees that is able to convert between the index of a node in a depth-first traversal (i.e. pre-order) and in a breadth-first traversal (i.e. level-order) of the tree. Using this, we are able to augment previous succinct data structures for interval graphs with the `distance` operation.

We also study several variations of the data structure problem in interval graphs: beer interval graphs and dynamic interval graphs. In beer interval graphs, we are given that some vertices of the graph are beer nodes (representing beer stores) and we consider only those paths that pass through at least one of these beer nodes. We give data structure results and prove space lower bounds for these graphs. We study dynamic interval graphs under several well known dynamic models such as incremental or offline, and we give data structures for each of these models.

Finally we consider path graphs where we improve on previous works by exploiting orthogonal range reporting data structures. For optimal space representations, we improve the run time of the queries, while for non-optimal space representations (but optimal query times), we reduce the space needed.

Acknowledgements

This thesis is the result of my work as Ph.D student at the University of Waterloo and would not have been possible without the support of my supervisor, Ian Munro. I have learned a lot under his supervision. I would like to also thank him for his feedback on drafts of this thesis.

I would like to thank my co-authors, for our many helpful discussions, research ideas and of course contributions to our papers.

I would like to thank my committee who have volunteered their time and offered their expertise. Thank you Prof. Lap Chi Lau, Prof. Richard Peng, Prof. Stephen Melczer and Prof. Get Brodal.

Lastly, I would like to thank my friends and family for their support, and motivation during my Ph.D which largely coincided with the Covid-19 pandemic.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As a result of the rapid growth of electronic data sets, memory requirements have become a bottleneck in many applications as performance usually drops dramatically as soon as data structures no longer fit into faster levels of the memory hierarchy in computer systems. To solve this problem, a class of data structures termed *succinct data structures* were proposed by Jacobson [49]. A succinct data structure uses information theoretic optimal space, plus lower order terms to represent combinatorial objects and perform the expected queries. Jacobson constructed such data structures for bit-vectors, trees and planar graphs. Since then many different objects have succinct data structures constructed supporting the relevant queries on them. For example:

- Strings [39]
- Dictionaries [72]
- Abelian groups [29]
- Permutations and functions [60]
- Many different classes of Graphs [30, 49, 24, 23, 17]

Although many of the results above are purely theoretical in nature, succinct data structures have found many practical applications as well. For instance many of the succinct data structures (such as bit-vectors, strings, suffix arrays, trees etc.) have been implemented in the Succinct Data Structures Library [38]. As an example of the space savings that can be achieved, using succinct trees to support the structure of XML documents uses only 3.12 to 3.81 bits per node [27], which is much smaller than the space required ($\Theta(\lg n)$ which would be 64 bits in a 64-bit architecture) in a traditional pointer based tree implementation.

This thesis will primarily focus on the space efficient representations of graphs, a versatile object representing objects and relationships between objects. In particular, we focus on graphs whose edges can be implicitly represented[1], so that even dense graphs with $\Theta(n^2)$ edges can be represented in close to linear space.

1.1 Organization of the Thesis

In Chapter 2, we give an overview of the background knowledge needed. We first introduce some standard definitions of the model of computation used and the objects we are studying. Next we give previous results on succinct data structures that we will use as building blocks for our results.

In Chapter 3 we give succinct data structures for ordinal trees which supports level-order (breadth-first traversal) operations in addition to the operations previously supported. The results will be a key building block for the data structures in subsequent chapters. This chapter is based on joint work with Meng He, Ian Munro, Yakov Nekrich and Sebastian Wild [43].

In Chapter 4 we study succinct static interval graphs and some related classes of graphs, such as proper interval graphs, circular arc graphs, and bounded degree interval graphs. We give succinct representations of these classes of graphs which supports navigational queries and distance queries. As mentioned, a key part of the data structures is the succinct level-order trees from Chapter 3 which allows space efficient implementation of the distance query. Chapter 4 is also based on joint work with Meng He, Ian Munro, Yakov Nekrich and Sebastian Wild [43].

In Chapter 5 we study the problem of supporting the beer distance query in interval graphs. In a beer graph, some of the vertices are designated as *beer vertices*, and in a beer distance query, the shortest path between two vertices must pass through one of the these special beer vertices. We give data structures for both interval graphs and proper interval graphs and via a counting argument prove a lower bound on the space of any data structure that supports this operation. This chapter is based on joint work with Ian Munro, Meng He, Rathish Das, Eitan Kondratovsky, and Anurag Murty Naredla [26].

[1]Edges are implicit in the sense that they are not given as pairs of vertices, but rather as the consequence of particular sets intersecting.

In Chapter 6 we study the problem of supporting the navigational and distance queries in dynamic interval graphs. In this setting, the interval graph changes by the insertion and deletion of vertices, as represented by an interval. The edges upon an insertion are exactly those that the new interval would intersect, and upon a deletion, we also delete all incident edges. We study the problem under several models of updates: no restrictions, allowing only insertions or deletions, or knowing all the operations in advance. This chapter is based on joint work with Meng He, Ian Munro, Richard Peng, Jingbang Chen, and Daniel Zhang.

In Chapter 7 we study path graphs and chordal graphs, two superclasses of graphs of interval graphs. We improve on previous results of succinct and compact data structures for them, with better query times for the navigational and distance queries. This chapter is based on joint work with Ian Munro and Meng He.

Chapter 2

Preliminaries

In this chapter we will introduce the necessary concepts, and results that will be needed in the rest of the thesis.

We will use the word-RAM model of computation [33] throughout the thesis, with word size $\omega = \Theta(\lg n)$ where n is the size of input.

2.1 Trees and Graphs

The objects we will study in this thesis are various classes of graphs, including trees. We begin by defining them.

Definition 2.1.1. A (undirected) ***graph*** G consists of a set of vertices $V(G)$ and a set of edges $E(G) \subset V(G)^2$, and if needed write $G = (V(G), E(G))$. When the graph is unambiguous, we will use V and E. We will use $n = |V|$ and $m = |E|$.

Definition 2.1.2. A graph G is ***weighted*** if we assign a real numbered weight to each edge. If all weights are 1, we say that the graph is ***unweighted***.

We will assume that our graphs are unweighted, so that each edge has weight 1, unless otherwise stated.

Definition 2.1.3. The operations we are interested in supporting in a graph data structure are:

- `adjacent`(u, v)[1]: given two vertices, are they adjacent (i.e. is $(u, v) \in E$)?
- `degree`(v): given a vertex, what is the number of vertices adjacent to it?
- `neighborhood`(v): given a vertex, list all vertices adjacent to it.
- `spath`(u, v): given two vertices, return a shortest (weighted) path between them.
- `distance`(u, v): given two vertices, return the length of a shortest path.

The operations below modify the graph. A graph data structure supporting these are *dynamic*.

- `insert`(v): add a new vertex v to the graph
- `insert`(u, v): given two vertices in the graph, add an edge between them.
- `delete`(v): delete the vertex v in the graph
- `delete`(u, v): delete the edge between u, v in the graph.

The main class of graphs we will study are *intersection graphs*, where edges are implicitly defined.

Definition 2.1.4. A graph G is an ***intersection graph*** if we may associate every vertex v with a set s_v such that for any two vertices, $(u, v) \in E$ if an only if $s_u \cap s_v \neq \emptyset$. We say that the family of sets is an ***intersection model*** for the graph.

Next we give the definition of a tree:

Definition 2.1.5. A rooted ***ordinal tree*** is an acyclic graph consisting of n vertices (which we will call *nodes*) and $n - 1$ edges, with one node designated as the root. All non-root vertices have a parent vertex, which is closer in distance to the root. The *children* of a node v are the nodes whose parent is v. A node is a *leaf* if it has no children. We note that for an ordinal tree, the order of the children of a node matters (i.e. two ordinal trees are isomorphic if for corresponding nodes, the children are the same and have the same order).

[1]Our data structures will have their own ways to referring of vertices, typically by giving each vertex a unique label between 1 and n, and the inputs to our operations will be the label of that vertex.

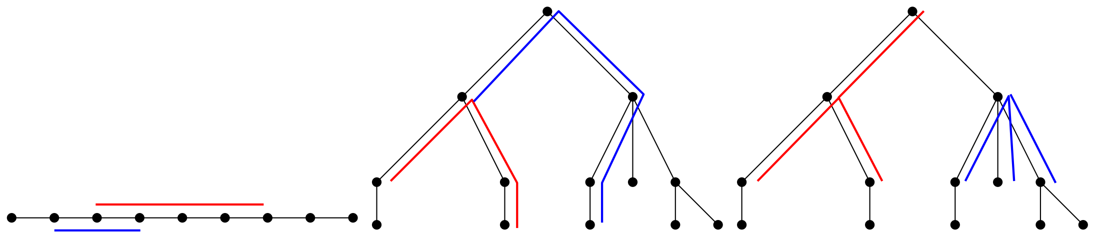


Figure 2.1: The sets of nodes for each of the types of intersection graphs.

.

Definition 2.1.6. A rooted ***cardinal tree*** (of cardinality k) is an ordinal tree where for every node, each child belongs to one of k slots, and no slot is used more than once.

Definition 2.1.7. A ***binary tree*** is an cardinal tree of cardinality 2, where we name the slots as "left" and "right". Thus, each child of a node v is either a left child or a right child. Consequently, if a node has a single child, then the 2 trees where the child is a left child versus a right child are different trees.

Lastly we will give the definition of the specific intersection graphs we will study:

Definition 2.1.8. A graph G is a ***chordal graph*** if it is the intersection graph of subtrees in a tree. That is, there exists an ordinal tree T, such that for every vertex $v \in V$, s_v is a set of connected nodes in T.

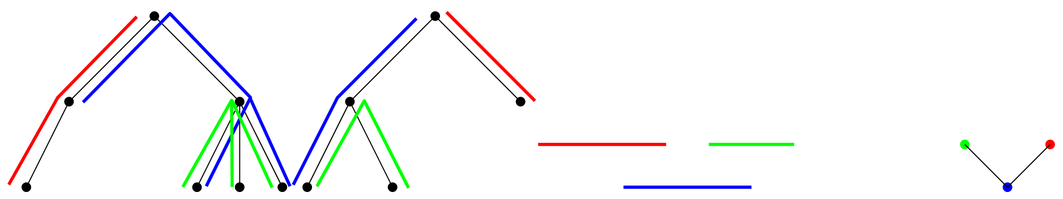
Definition 2.1.9. A graph G is a ***path graph*** [2] if it is the intersection graph of (simple) paths in a tree. That is, there exists an ordinal tree T, such that for every vertex v, s_v is a set containing the nodes of a simple path in T.

Definition 2.1.10. A graph G is an ***interval graph*** if it is the intersection graph of (simple) paths in a path. That is, there exists an ordinal tree T that is a path, such that for every vertex v, s_v is a set containing the nodes of a simple path in T.

From these definitions, it is easy to see that the set of graphs form the following inclusion relationship: interval graph $\subseteq$ path graph $\subseteq$ chordal graph.

[2] The object we are defining is not a path, which is what is often referred to by the term path graph.

Example 2.1.11. The following illustrates a graph (right), and the 3 types of sets corresponding to that of a chordal graph, path graph and interval graph, that create this graph as their intersection structure.



2.2 Succinct Data Structure

The main topic of this thesis is space-efficient data structures. Given set of combinatorial objects X[3], the information theory lower bound states that to represent the objects in X, some object must use at least $\lceil \lg |X| \rceil$ bits. In other words, any data structure for objects in X must use at least $\lceil \lg |X| \rceil$ bits in the worst case. To see this, if we use fewer number of bits, then as we have fewer bit strings than number of objects, two objects must have the same bit-string as their data structure, but clearly they cannot since some query must differ for different objects, but with the same data structure, all queries would be the same.

Definition 2.2.1. A data structure for a class of combinatorial objects X is ***succinct*** if it uses $\lg |X| + o(\lg |X|)$ bits in the worst case. It is ***compact*** if it uses $\Theta(\lg |X|)$ bits in the worst case.

The data structure will naturally need to support the relevant queries on X. [4]

The information theoretic lower bound is often obtained by counting the number of objects X. For example, if X is the set of ordinal trees (Definition 2.1.5) with n vertices, then $\lg |X| = 2n - o(n)$ [79]. If X is the set of all graphs (Definition 2.1.1), then $\lg |X| = \frac{1}{2}n^2 - o(n^2)$. [5]

[3]Two objects $x, y \in X$ are the same if you cannot distinguish them from the queries supported on the objects. For example, `adjacent` in a graph.

[4]One can naively represent the objects by simply enumerating them and giving them numbers $1, \ldots, |X|$. Such a scheme is not a *data structure* since we cannot answer queries - such as `adjacent` for a graph.

[5]Each of the $\binom{n}{2}$ possible edges is either there or not. Since each graph has at most $n!$ automorphisms, we over count by a factor of at most $n!$. Giving a rough count of $2^{\binom{n}{2}}/n!$

2.3 Simple Succinct Data Structures

Here we give previous results on some simple data structure that we will use as building blocks throughout the thesis.

First is a bit-vector.

Definition 2.3.1. A ***bit-vector*** of length n is an array of n bits. It supports the following operations:

- $\texttt{access}(i)$: return the bit at index i.
- $\texttt{rank}_b(i)$: return the number of "b" bits at or before the index i, where b is either 0 or 1. If we omit b, it is assumed that we mean $b = 1$.
- $\texttt{select}_b(i)$: return the index of the i-th "b" bit. If we omit b, it is assumed that we mean $b = 1$.

Lemma 2.3.2 ([62]). *A bit-vector of length n can be represented in $n + o(n)$ bits to support* $\texttt{access}, \texttt{rank}, \texttt{select}$ *in $O(1)$ time.*

Example 2.3.3. Consider the following bit-vector B.

0	1	0	1	0	1	1	1	0	1	1

$\texttt{access}(B, 3) = 0$ as the third bit is 0. $\texttt{rank}(B, 5) = 2$ since there are 2 1 bits in the prefix $[1, \ldots, 5]$. Lastly $\texttt{select}(B, 6) = 10$ as the index of the 6th 1 bit is 10.

When the number of "1" bits is small (i.e. $o(n)$ or symmetrically is large i.e. $n - o(n)$), the number of bit-vectors with that many 1s is also smaller, and we can use fewer bits to represent the bit-vector. These *compressed* bit-vectors are a generalization of the above.

Lemma 2.3.4 ([69]). *For a bit-vector of length n, containing m 1-bits*[6]*, and any constant c, there is a data structure using*

$$\lg \binom{n}{m} + O\left(\frac{n}{\lg^c n}\right) \leq m \lg \left(\frac{n}{m}\right) + O\left(\frac{n}{\lg^c n} + m\right)$$

bits of space that supports $\texttt{access}, \texttt{rank}, \texttt{select}$ *in $O(1)$ time.*

[6]The number of such bit-vectors is $\binom{n}{m}$.

Next we generalize the bit-vector to larger alphabets.

Definition 2.3.5. A ***string*** S of length n on an alphabet $\Sigma = \{1, \ldots, \sigma\}$ is a sequence n characters belonging to Σ. The operations generalize naturally as:

- `access`(i): return the character at index i.
- `rank`(c, i): return the number of occurrences of the character c in the prefix up to (and including) the index i.
- `select`(c, i): return the index of the i-th occurrence of the character c.

The result we will use is.

Lemma 2.3.6 ([39]). *A string S of length n over an alphabet of size σ, can be succinctly represented using $n \lg \sigma + o(n \lg \sigma)$ bits to support* `rank`, `access` *in $O(\lg \lg \sigma)$ time and* `select` *in $O(1)$ time.*

Last of the simple data structures we will use is permutations.

Definition 2.3.7. A ***permutation*** P of size n is a bijective function from $\{1, \ldots, n\}$ to itself. The two operations are:

- $P[i]$: return the value that i is sent to by the function.
- $P^{-1}[i]$: return the value that is sent to i by the function.

Lemma 2.3.8 ([60]). *Let P be a permutation. Then we may represent P using $(1 + 1/f(n))n \lg n + o(n \lg n)$ bits to support the computation of P and P^{-1} in $O(f(n))$ time ($1 \leq f(n) \leq n$). In particular, if we set $1/f(n) = \varepsilon$ for constant $\varepsilon > 0$, then the space is $(1 + \varepsilon)n \lg n + o(n \lg n)$ bits and the time is $O(\frac{1}{\varepsilon}) = O(1)$.*

2.4 Trees

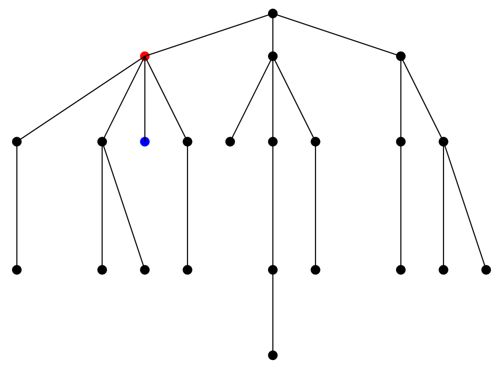
As trees are a fundamental object in computer science, succinct trees have been intensively studied. In Chapter 3 we will make our own contribution in the study of succinct trees. For now we will describe the operations that succinct trees are able to support, see Table 2.1.

`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 ($i \in \{1, \ldots, \texttt{degree}(v)\}$)
`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 2.1: Navigational operations on succinct ordinal trees. (v denotes a node and i an integer). The operations highlighted are not supported in previous results, and are studied in Chapter 3.

Theorem 2.4.1 ([31]). *An ordinal tree can be represented in $2n + o(n)$ bits and can support all the operations of Table 2.1 with the exception of* `prev_internal`, `next_internal`, `node_rank`$_{\texttt{LEVEL}}$, `node_select`$_{\texttt{LEVEL}}$ *in $O(1)$ time.*

Example 2.4.2. Consider the following ordinal tree.



In a preorder traversal of the tree, the red node is the second node in the traversal and the blue node is the 8th node in the traversal. Therefore, $\texttt{node\_select}_{\texttt{PRE}}(2)$ would return some representation of the red node, which we can then use as input to the other operations, which typically is the preorder rank of the node: 2. Using this as our representation, $\texttt{degree}(2) = 3$.

Closely related to the lowest common ancestor (LCA) operation is the range-minimum and range-maximum queries.

Definition 2.4.3. Let A be an array of numbers. A ***range-minimum*** query is the following:

- Given two indices i, j, return the index k, $i \leq k \leq j$ such that $A[k] = \min_{i \leq k \leq j} A[k]$ (i.e. the index between i and j containing the minimum element in that range)

By constructing a Cartesian tree [82], where we take the minimum value of the array of the root and recursing on both sides, the range-minimum query is equivalent to the LCA query. The node with the minimum value between i and j has the lowest depth in the Cartesian tree between then nodes at indices i, j in an in-order traversal.

Thus we have from Theorem 2.4.1:

Lemma 2.4.4. *We may solve the range-minimum problem in $2n + o(n)$ bits and $O(1)$ time.*

Symmetrically, we may define and solve the ***range-maximum*** problem.

2.5 Orthogonal Range Reporting

Next we describe some more complex building blocks that we will need.

The data structure stores n d-dimensional points, and we wish to answer queries based on the points inside d-dimensional axis aligned rectangles.

Definition 2.5.1. Let S be a set of n d-dimensional points (with polynomial sized coordinates). Let R be a d-dimensional axis aligned rectangle (given as input to a query): $[p_1, p_2] \times [p_3, p_4] \ldots [p_{2d-1}, p_{2d}]$ [7]. The queries we wish to support are:

- *emptiness*: is there a point of S inside R?
- *count*: how many points of S are inside R.
- *reporting*: return every point of S inside R.

The rectangle R is k-sided if there is at most k coordinates among p_i that are finite.

It is clear that a data structure that only supports k-sided queries will be easier than if we do not have this restriction. When we do not mention how many sides, it is assumed that it is the maximum possible: $2d$.

First we consider the problem in 2 dimensions.

Lemma 2.5.2 ([20])**.** *We can solve the 2D range emptiness and reporting queries using $O(n \lg n \lg\lg n)$ bits of space and $O(\lg\lg n)$ query time (for the reporting query, this is per point in the output) or $O(n \lg n)$ bits of space and $O(\lg^{\epsilon} n)$ query time for any constant $\epsilon > 0$.*

If we need the space to be exactly $n \lg n + o(n \lg n)$ bits, we have:

Lemma 2.5.3 ([16])**.** *Let S be a set of points from the universe $M = [1..n] \times [1..n]$, where $n = |S|$. S can be represented using $n \lg n + o(n \lg n)$ bits to support orthogonal range counting in $O(\lg n / \lg\lg n)$ time, and orthogonal range reporting in $O(k \lg n / \lg\lg n)$ time, where k is the size of the output.*

The standard way of generalizing the above restriction on the points to the region $[1..n] \times [1..n]$ is to use a predecessor data structure:

[7] We use closed interval here, but it is easy to see how to convert these into open/semi-open intervals when the coordinates are integral

Definition 2.5.4. The data structure stores a set of n numbers S out of a universe $U = [1, |U|]$. The queries are:

- `pred`(i), given an element i of U, return the largest element j of S that is smaller than i.
- `succ`(i), given an element i of U, return the smallest element j of S that is larger than i.

First we note that if $|U|$ is small, for example $|U| = O(n)$, then we may solve this with a bit-vector, where a 1 at index i states that $i \in S$. `pred` and `succ` can be reduced to `rank` and `select`, the space would be $|U| + o(|U|) = O(n)$ bits with constant time complexity. For larger universes ($|U|$ =poly(n)), though there have been a lot of work on this problem, we will only need one of the basic results by Willard [83] which gives a $O(n)$ word space solution to the problem with $O(\lg\lg|U|)$ query time.

Lemma 2.5.5 ([83])**.** *There is a data structure for the predecessor/successor problem that uses $O(n)$ words of space and query time $O(\lg\lg|U|)$.*

By combining the above, if the points' coordinates stored in Lemma 2.5.3 are $O(n)$, then we convert them into *rank-space* (i.e. the index the point would have in sorted order) before storing the points. When we return the points, we convert them back from rank-space.

Lastly we consider the 2-dimensional 3-sided reporting query problem. This problem has a folklore solution that we will leverage in this thesis. As we will use this frequently, we will describe the data structure in detail.

Let S be a set of n points on a 2D plane, and $R = [x_1, x_2] \times [y_1, y_2]$ be an axis aligned rectangle. By assumption, one of x_i, y_i is $\pm\infty$.

Without loss of generality, suppose that $y_2 = \infty$. We solve the 3-sided range reporting query in the following way: first convert x_1, x_2 into rank-space. That is rank$(x_1) = r_1$ where there are r_1 points with x-coordinates smaller than x_1; similarly for x_2. Store a range maximum data structure (Lemma 2.4.4) such that given two ranks r_1, r_2, it returns the index (rank) of the point in this range with the maximum y-coordinate - denote this by r'. If the y-coordinate of this point is greater than y_1, return the point and recurse on the two sub ranges $[r_1, r')$ and $(r', r_2]$.

If $y_1 = -\infty$ instead, we only need to replace the range maximum data structure with a range minimum data structure. If either $x_1, x_2 = \pm\infty$, then it is symmetric and we need to be able to compute the rank using y-coordinates.

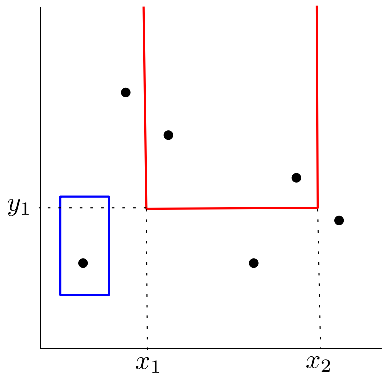


Figure 2.2: 2-dimensional range search: the red rectangle would correspond to $[x_1, x_2] \times [y_1, \infty]$, and is 3-sided. A reporting query would return the 2 points in it, an emptiness query would return false and a counting query would return 2. The blue rectangle is 4-sided, and would return the single point contained in it.

Thus to support the 2-dimensional 3-sided queries, we must be able to support the following operation.

- given an x-coordinate convert it to a rank - which we will denote as `rank`.
- given the rank of the x-coordinate of a point, compute both the x and y-coordinate of the point - which we will denote as `decode`.

Aside from these tasks, we will need to store a range minimum/maximum data structure, which uses $2n + o(n)$ bits by Lemma 2.4.4.

The first task, `rank` is typically done through a predecessor data structure, though if the universe of the points is small, say $x = O(n)$ we can use a bit vector instead to save on space.

The second task, `decode` is typically done by storing the points explicitly using $n \lg n$ bits in sorted order by the x-coordinate. Thus to find the y-coordinate of the k-th point, we can simply look at the k-th entry in the array. However, if there are many such range reporting data structures, then we only need to store this array once. Furthermore, if the y-coordinate is somehow easily computable from the x-coordinate, then we only need to store the function computing the y coordinate once, and save on space.

The time complexity is $O(f + k \cdot g)$ where k is the number of points returned, f is the time to convert x coordinates to ranks and g is the time to compute the y-coordinate given the rank of a point.

We may summarize result as:

Lemma 2.5.6. *Given n 2-dimensional points, and let f be the time cost of* `rank`*, g be the time cost of* `decode`*. Then we are able to support 3-sided reporting queries (of the form $[x_1, x_2] \times [-\infty, y]$, to support the symmetric cases, we duplicate the structure) in time $O(f + k \cdot g)$ where k is the number of points reported. The space cost is $2n + o(n)$ plus the space needed to support the* `rank` *and* `decode` *operations.*

Lastly, we move to 3-dimensions and 5-sided rectangles. The following result is by Nekrich [68].

Lemma 2.5.7 ([68])**.** *For n 3D points and constant $\varepsilon > 0$, we may support 5-sided orthogonal reporting queries using $O(n \lg n)$ bits of space and $O(k \lg^{\varepsilon} n)$ time or $O(n \lg n \lg \lg n)$ bits of space and $O(k \lg \lg n)$ time, where k is the size of the output.*

We may also achieve the same complexities for emptiness as well.

Proof. The original result did not have the emptiness query. However there is a well-known black box reduction to emptiness queries from reporting. Suppose that the run time of the query is bounded by $t_1 + kt_2$, then we may run the query for time t_1 and if it has not returned, answer no, and otherwise no points are returned and we may answer yes.

In this case $t_1 = t_2 = c \lg^{\epsilon} n$ or $c \lg \lg n$, for some suitably large constant c. $\square$

Chapter 3

Level-Order Succinct Trees

In this chapter, we will study succinct ordinal trees which are able to support level-order operations. These trees will be a key building block for the data structures we will construct in later chapters.

This chapter is organized as follows: we begin with an introduction in Section 3.1, followed by a overview of the vast amount of previous work on constructing succinct tree data structures in Section 3.2. Next we give the terminology used and other succinct data structures we will use as our building blocks in Section 3.3. We give the construction of our data structure and how we support level-order operations in Section 3.4, and how we support all the operations previously supported in Section 3.5. Finally we summarize our results and discuss further avenues of research in Section 3.6.

3.1 Introduction

Standard pointer-based representations of trees use $\Theta(n)$ words or $\Theta(n \lg n)$ bits to represent a tree on n nodes, but as the culmination of extensive work [49, 25, 61, 62, 23, 63, 56, 74, 67, 12, 50, 10, 37, 44, 32], ordinal trees can be represented *succinctly*, using the optimal $2n + o(n)$ bits of space, while supporting a plethora of navigational operations in constant time, see Table 2.1.

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`) [49] – and very limited in terms of supported operations, or they are depth first – like the *depth-first unary degree sequence* (DFUDS) [12], the *balanced-parentheses* (BP) encoding [61] and *tree covering* (TC) [37] – and do not support level-order ranks.

In this chapter, we construct a tree data structure that bridges the dichotomy, solving an open problem of [44][1]. 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.

3.2 Previous Work

As trees are a fundamental data structure in computer science, there has been intense study of succinct trees in the literature. Many different approaches have been used, each with its own strengths and weaknesses, such as the variety of operations supported, whether they support dynamic operations (such as inserting and deleting nodes). We give an overview of the history of the study of succinct trees.

The *level-order unary degree sequence* (LOUDS) representation of an ordinal tree [49] 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 [6], but a major disadvantage of LOUDS-based data structures is that they support only a very limited set of operations [66].

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 [12]. Other approaches that allow to support largely the same set of operations are based on the *balanced-parentheses* (BP) encoding [61] or rely on *tree covering* (TC) [37] for a hierarchical tree decomposition.

[1]The problem of supporting rank/select operations on the level-order traversal of the tree, while at the same time, supporting all other previous operations.

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 [61] 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. [62] 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. [23] showed how to support degree using the BP representation in constant time which is needed for succinct graph representations, while Munro and Rao [63] 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 [56], that for $\texttt{node\_rank}_{\texttt{IN}}$ and $\texttt{node\_select}_{\texttt{IN}}$ by Sadakane [74] in their work of encoding suffix trees, and that for level_leftmost and level_rightmost by Navarro and Sadakane [67].

Benoit et al. [12] 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. [50] 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 labelled trees, Barbay et al. [10] 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. [37] 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. [44] further showed how to use TC to support all other operations provided by BP and DFUDS representations in constant time. Later, based on a different tree covering algorithm, Farzan and Munro [32] designed 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.

3.3 Preliminaries

In this chapter, we will require several results stated in Chapter 2. First are the various bit-vector results:

Lemma 2.3.2 ([62]). *A bit-vector of length n can be represented in $n + o(n)$ bits to support* access, rank, select *in $O(1)$ time.*

Lemma 2.3.4 ([69])**.** *For a bit-vector of length n, containing m 1-bits*[2]*, and any constant c, there is a data structure using*

$$\lg \binom{n}{m} + O\left(\frac{n}{\lg^c n}\right) \leq m \lg \binom{n}{m} + O\left(\frac{n}{\lg^c n} + m\right)$$

bits of space that supports `access`, `rank`, `select` *in $O(1)$ time.*

Using these we have the following notion:

Definition 3.3.1. Let A be an array storing values of k bits. We say that A is piece-wise linear with M pieces if there are M runs of identical values.

Lemma 3.3.2. *Suppose that A is a piece-wise linear array of length N with M pieces storing values of k bits with $k = O(c \lg n)$ for some constant c (i.e at most a constant number of words).*

Suppose that $M = n/f(n)$, $N = n \lg^{c_1} n$ and $k = o(f(n))$ for $f(n) = \Omega((\lg \lg n)^{c_2})$ and constants c_1, c_2. Then we may store A in $o(n)$ bits and access $A[i]$ in $O(1)$ time if $c_2 > 1$.

Proof. We store a bit-vector B of length N with M 1s where $B[i] = 1$ if i is the first index of a run in A.

Store B using a compressed bit vector Lemma 2.3.4. The space required is $M \lg N/M + O(n/\lg^c n + M)$. By our assumption, this is $\frac{n}{f(n)} \lg((c_1 \lg n) f(n)) + O(n \lg^{c_1 - c} n + n/f(n))$ bits of space. For this to be $o(n)$ we choose $c > c_1$ and $c_2 > 1$ so that $\frac{\lg(c_1 \lg n) + \lg f(n)}{f(n)} = o(1)$.

We store the value of each run of A in a size M array, with each slot using k bits. By assumption this is $o(n)$ bits. To find $A[i]$, we use $\texttt{rank}_B(i)$ to find the run number, then find the value of that run. $\square$

In this chapter, we will be augmenting the previous succinct tree data structures with level-order based operations. The previous data structure we will augment is:

Theorem 2.4.1 ([31])**.** *An ordinal tree can be represented in $2n + o(n)$ bits and can support all the operations of Table 2.1 with the exception of* `prev_internal`, `next_internal`, $\texttt{node\_rank}_{\texttt{LEVEL}}$, $\texttt{node\_select}_{\texttt{LEVEL}}$ *in $O(1)$ time.*

The additional operations we will add are `prev_internal`, `next_internal`, $\texttt{node\_rank}_{\texttt{LEVEL}}$ and $\texttt{node\_select}_{\texttt{LEVEL}}$, so that we may support all the operations in Table 2.1 in constant time.

[2]The number of such bit-vectors is $\binom{n}{m}$.

3.4 Main Data Structure

In this section, we describe the construction of the main data structure for the ordinal tree. As in the previous works for ordinal trees based on tree covering, the main data structure describes the decomposition of the tree and uses $2n + o(n)$ bits. For each tree operation or query, we construct auxiliary data structures occupying $o(n)$ bits which allows us to answer the query in $O(1)$ time[3].

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 (i.e. $\texttt{node\_rank}_{\texttt{PRE}}$). Tree covering relies on a two-tier decomposition: the tree consists of mini-trees, each of which consists of micro-trees. The mini-trees will be denoted by μ^i, the micro-trees (belonging to μ^i) by μ^i_j.

We will build upon previously used tree covering decomposition schemes. A greedy bottom-up approach suffices to break a tree of n nodes into $O(n/B)$ subtrees of $O(B)$ nodes each [37]. However, more carefully designed procedures yield restrictions on the touching points of subtrees:

Lemma 3.4.1 (Tree Covering, [31, 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.*

1. *Subtrees are pairwise disjoint except for (potentially) sharing a common subtree root.*
2. *Each subtree contain at most $2B$ nodes.*
3. *The overall number of subtrees is $\Theta(n/B)$.*
4. *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. The case where a subtree contains two blocks arises when the subtree root has exactly one heavy child: a node whose subtree size is greater than B, in the decomposition algorithm.

[3]In the following lemmas in this chapter, we will not explicitly state that the run time of the operations are $O(1)$.

The main feature of this decomposition algorithm is that there is at most one external edge in each subtree, which allows us to compute information about the subtree (which uses the external edge) in constant time without more careful work.

Now we ask the question: why are level-order operations hard? As discussed, the only data structure implementing level-order operations is LOUDS [49], which does not support any operations at all outside of local navigational operations such as `parent` and `child`. First consider the naive approach: 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). To see that we need to store the level-order ranks for every leftmost node in μ^i, we note that the level-order traversal enters some level of μ^i, then leaves it. But the number nodes between leaving μ^i and re-entering μ^i on the next level is arbitrary, and without storing this information, we cannot calculate the ranks of a node v in μ^i from only the information of the levels above v.

The problem however is that, for general ordinal trees, we cannot afford to store the level-order rank of all leftmost nodes. This would require $\texttt{height}(\mu^i) \cdot \lg n$ bits for $\texttt{height}(\mu^i)$ 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 $|\mu^i| = \omega(\texttt{height}(\mu^i) \lg n)$ nodes or $\texttt{height}(\mu^i) = o(|\mu^i|/\lg n)$. 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.

3.4.1 Covering by Slabs

To bound the height of the subtrees, we first perform a horizontal partitioning step. We fix two parameters: $\boldsymbol{H}$, the height of slabs, and $\boldsymbol{B} > H$, 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 can 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.[4]

We will identify induced subgraphs with the set of nodes that they are induced by. So $S_i = \big\{v : \mathtt{depth}(v) \in [(i-1)H + o \ \ .. \ \ iH + o]\big\}$, 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 $h + 1 \leq n/H + 2$. 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. 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 $|S_i| \leq B$, 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 3.4.1) 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 3.1.

Theorem 3.4.2 (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] To see this, over all offsets o, every node is a slabbed node for exactly 1 offset. Thus the sum over all offsets of the slabbed nodes is n. By pigeonhole principle, for some offset, the number of slabbed nodes is at most the average number of slabbed nodes n/H.

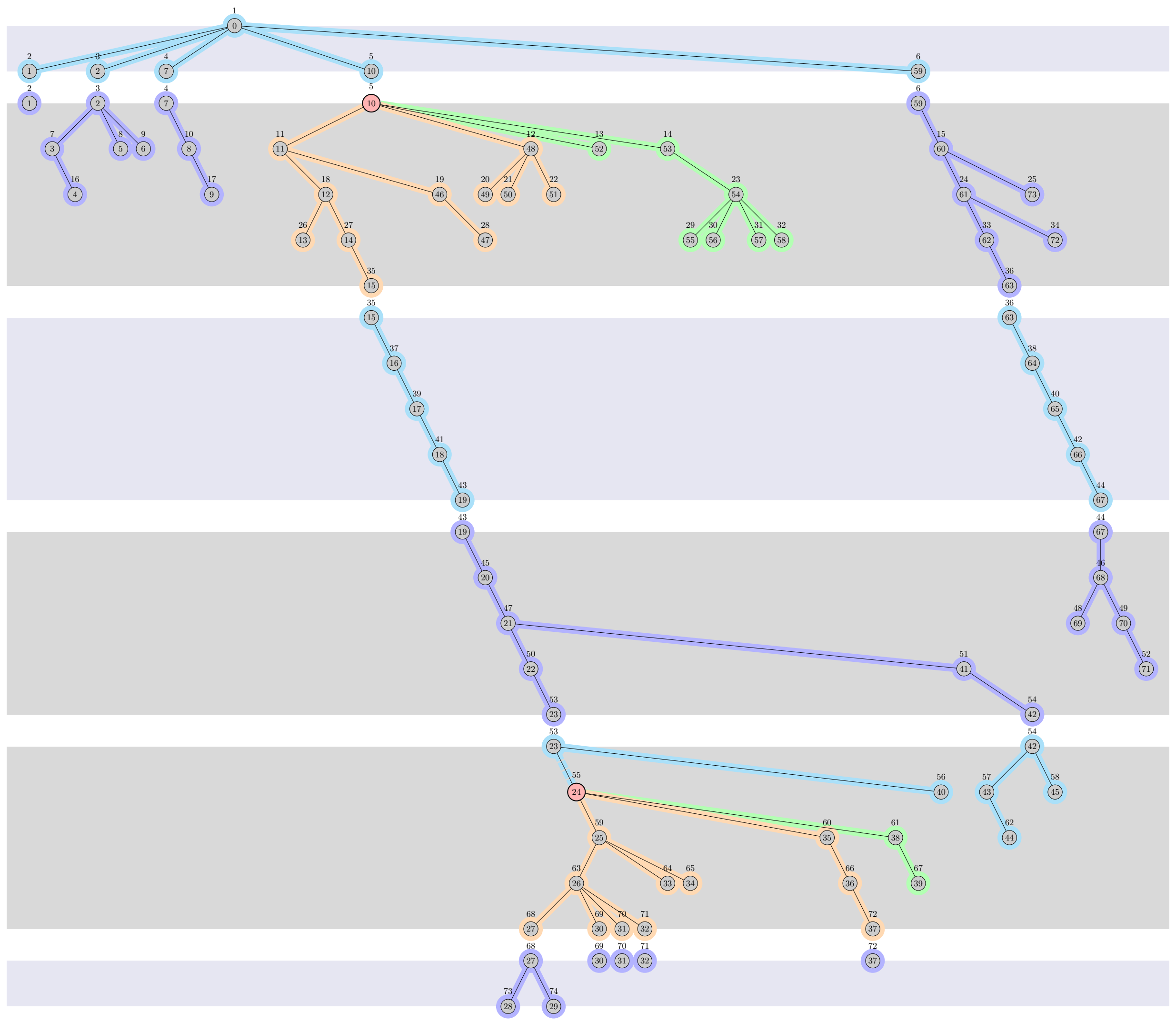


Figure 3.1: An example of the tree-slabbing decomposition from Theorem 3.4.2 with $B = 11$ and $H = 4$. Slabs are shown as shaded areas (light blue for skinny slabs, light gray for fat slabs). All s-nodes are depicted twice, one in each slab they belong to. The trees within a slab are connected by a dummy root (not depicted) and further decomposed as in Lemma 3.4.1; the resulting subtrees are shown by the edge colors.

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

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 $m = O(n/H) = O(n/\lg^3 n)$, only $O(n/M) = O(n/\lg^5 n)$ 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.

Definition 3.4.3. The parameters for our tree-slabbing scheme are $H = \lceil \lg^3 n \rceil$ and $B = \lceil \lg^5 n \rceil$ and $H' = \lceil \frac{\lg n}{(\lg \lg n)^2} \rceil$ and $B' = \lceil \frac{1}{8} \lg n \rceil$

After these two levels of recursion we have reached a size for micro trees small enough to use a lookup table that takes sublinear space. That is, for every possible micro tree, and every possible query on the micro tree (such as depth of a node) we store the result of the query. Since the size of the micro trees is at most $\frac{1}{4} \lg n$, the number of such trees is $O(\sqrt{n})$. If a query takes as input k nodes, and outputs an answer of size $O(\lg n)$, then the space cost of storing that query would be $O(\sqrt{n} \lg^{k+1} n) = o(n)$ bits. Over a constant number of queries, the total space cost would be $o(n)$.

As in previous tree covering data structures, 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 $\mu^{\tau_1}_{\tau_2}$; 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.

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 [31] 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.

3.4.2 Internal Data Structure Operations

We begin with some common concepts and operations internal to our data structure that we will use, and will be helpful in the implementation of many queries.

The *type* of a micro tree is the concatenation of its size, a description of its local shape (such as a balanced parenthesis sequence), and the preorder rank of the promoted dummy node corresponding to the external edge (0 if there is none), and several bits indicating whether the lowest level are promoted s-nodes, whether the root is a dummy root, whether the tree was obtained by composing a fat slab or a skinny subtree and so on.

We store a variable-cell array of the *types* of all micro trees in τ-order. The balanced parenthesis sequence 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 $O(\sqrt{n}\texttt{polylog}(n))$ bits.

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.

Definition 3.4.4. A node v so that $\tau_1(v) \neq \tau_1(v-1)$ is called a ***(tier-1) preorder changer*** [44, 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 3.4.2, 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 2.3.4) to indicate which nodes in a preorder traversal are (tier-1/tier-2) preorder changers. The space for that is $O(\frac{n}{H}\lg H + n\frac{\lg\lg n}{\lg n}) = o(n)$ for tier 1 and $O(\frac{n}{H'}\lg H' + n\frac{\lg\lg n}{\lg n}) = O(n\frac{(\lg\lg n)^3}{\lg n}) = o(n)$ for tier 2.

Thus given a vertex v by its pre-order number, we may find the tier-1/tier-2 preorder changer associated with v by selecting the first 1 in the appropriate bit vector preceding the index of v.

Next suppose that v is given by its τ-name, we wish to find the tier-1/tier-2 preorder changers (by their τ-name) associated with v.

First consider the tier-2 pre-order changer u'. By definition of a preorder changer, we have that $\tau_1(u') = \tau_1(v)$ and $\tau_2(u') = \tau_2(v)$. The node in the micro-tree that is on the boundary (i.e external edge, blocks of the children of the root, or slabbed edges) can be precomputed in the micro-tree lookup table, and this is the tier-2 preorder changer. We note that tier-1 preorder changes are by definition tier-2 preorder changers as well, thus the tier-1 preorder changer u cannot be between v and u' and thus u is the tier-1 preorder changer associated with both v and u'. To find u, we find tier-1 preorder changer associated with u'. We may do this by explicitly storing τ_2 and τ_3 (i.e. the pair $\langle\tau_2, \tau_3\rangle$) values for u since it would be $O(\lg\lg n)$ bits.

Thus the array containing the tier-1 and tier-2 preorder changers associated with each node in τ-order is piece-wise linear, of length $n \cdot \texttt{polylog}(n)$, with at most n/H' pieces in the case of tier-2 changers. Therefore we may store these using Lemma 3.3.2 in $o(n)$ bits. As in the parameter in Lemma 3.3.2, we can afford to store $O(\lg n)$ bits for each tier-1 changer and $O(\lg\lg n)$ bits for each tier-2 changer in an array.

In this way we have the following internal operation:

Lemma 3.4.5. *We can support the following operation: given a node v in either preorder numbers or τ-name, we are able to lookup any value associated with v's tier-1 and tier-2 preorder changer, of size $O(\lg n)$ for tier-1 and of size $O(\lg \lg)$ for tier-2, using $o(n)$ extra bits of space.*

We note that we were able to do this because the number of preorder changers is relatively small so that we are able to store a compressed bit-vector for them, and secondly, the tier-2 preorder changer can be determined from the micro-tree alone.

In the same vein, the analogous result can be extended to postorder, inorder etc. as well.

Now we consider the analogous result for level-order traversals.

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 have the definition of a level-order changer

Definition 3.4.6. 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 3.4.7. *The number of tier-1 (tier-2) level-order changers is $O(n/H + nH/B) = O(n/\lg^2 n)$ $(O(n/H' + nH'/B') = O(n/(\lg\lg n)^2))$.*

Proof. We focus on tier 1; tier 2 is similar. Lemma 3.4.2 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.

The only corner case is for tier-2, since a skinny micro-tree decomposed from a fat mini-tree does not contribute a single level-order changer (but a skinny micro-tree decomposed from a skinny mini-tree still contributes a single one) but rather one for each of its levels. However, in this case, every tier-2 changer is also a tier-1 changer since the boundary of the micro-tree (the left most nodes on each level) is also the boundary of the mini-tree, and thus the number of such tier-2 changers is negligible. □

Furthermore, the tier-2 level-order changer can be found from the micro-tree alone.

Therefore, we have the following result on this internal operation:

Lemma 3.4.8. *Given a node v by its place in X-order traversal or by τ-name, where X is* PRE, POST, LEVEL, DFUDS, IN*, we may find its tier-1/tier-2 X-order changer and also look up any information associated with its tier-1/tier-2 X-order changer of size $O(\lg n)$ bits for tier-1 and $O(\lg \lg n)$ bits for tier-2.*

Next we will show that we are able to store information for the mini-trees and micro-trees. For the mini-trees and micro-trees in τ order we store an array, indexed by the τ-names of the mini-trees and micro-trees: for a mini-trees, its τ_1 and for micro-tree its $\langle \tau_1, \tau_2 \rangle$. Since we have at most n/H mini-trees and n/H' micro-trees, we are able to store $O(\lg^2 n)$ bits per mini-tree and $O((\lg \lg n)^c)$ bits per micro-tree while keeping the space a lower order term. Thus we have the following lemma:

Lemma 3.4.9. *Let v be a node given by τ-name, then we can retrieve any information related to its mini-tree and micro-tree as long as it is $O(\lg^2 n)$ bits for mini-trees and $O((\lg \lg n)^c)$ bits for micro-trees. The space requirement is $o(n)$.*

Next we give some operations related to slabbed nodes. First, given a tier-1 slabbed node, we want its rank among tier-1 slabbed nodes in a level-order traversal. Note that since slabbed nodes are promoted, each one is traversed twice.

Let v be given by its τ-name. The first task we would like to accomplish is given a tier-1/tier-2 slabbed node by τ-name, find its rank in level-order among all tier-1/tier-2 slabbed nodes. If v is a promoted s-node, then it is on the last level of both its mini-tree and micro-tree. If v is an original s-node then it is on the first (after the dummy root) level of both its mini-tree and micro-tree.

Let u' be the closest node in level-order to the left of v such that its predecessor is either not a s-node or belongs to a different micro-tree. That is, in the micro-tree, u' is on the same level as v and is either a tier-2 level-order changer, or is the left most node of that level in the case that it is a skinny subtree and v is promoted so that it is on the last level of the micro-tree. We store this distance in the micro-tree lookup table. We note that for any such micro-tree, there is a constant number of possible u' since the number of level-order changers is constant and there is 1 extra node that maybe possible. For each

u', let u be the analogous node in the mini-tree. By the same analysis, for each mini-tree, there is a constant number of such possible u. At each micro-tree and for each u', we store $\langle \tau_2, \tau_3 \rangle$ for u and the number of nodes between u' and u. For each mini-tree, and for each possible u in the mini-tree, we store the level-order rank of u among s-nodes. Thus the level-order rank of v among all tier-1 s-nodes can be calculated as $u + (u' - u) + (v - u')$ since every node on this level are s-nodes. Here we have abused notation and denoted u as the level-order rank of u among s-nodes. By Lemma 3.4.9 we can store this information in $o(n)$ bits.

Next let v be a tier-2 s-node. We traverse the nodes first by mini-tree (τ_1), then traverse each mini-tree in level-order. In this way we have an ordering on tier-2 s-nodes. We wish to find the rank of v in this order.

Let v be given by its τ-name be a tier-2 s-node. Again, let u' be the closest node in level-order to the left of v such that its predecessor is either not a s-node or belongs to a different micro-tree. Let u be the first tier-2 s-node in level order in the mini-tree τ_1. We store the distance between u' and v in the micro-tree lookup table. For each u' in the micro-tree, we store the number of s-nodes between u' and u. For each mini-tree, we store the rank of u. Thus again we may find the rank of v as $u + (u' - u) + (v - u')$. By Lemma 3.4.9, we may do this in $o(n)$ bits.

Thus we obtain the following lemma:

Lemma 3.4.10. *Let v be a tier-1/tier-2 s-node. We may find the rank of v among tier-1/tier-2 s-nodes.*

This allows us to perform the following:

Corollary 3.4.11. *Let v be a tier-1/tier-2 s-node given by τ-name. We may lookup any information associated with v as long as the information is $O(\lg n)$ bits for tier-1 and $O(\lg\lg n)$ bits for tier-2.*[5]

Proof. We store an array in the order of the s-nodes. Given v, we convert it into a rank and index that rank in the array. For the array to be $o(n)$ bits we can only store $O(\lg n)$ and $O(\lg\lg n)$ bits respectively ☐

Some immediate application is the following: given a promoted s-node by τ-name, we may find the τ-name of the original s-node and vice versa.

[5]Can even be $O(\lg^2 n)$ and $O((\lg\lg n)^c)$ bits while keeping the total space $o(n)$ bits.

Next we wish to support the operations for finding s-node ancestors and s-node descendants of nodes.

Let v be a node given by τ-name. We wish to find u' the closest tier-1/tier-2 s-node ancestor of v if they exist.

For tier-2, it is either in the same micro-tree (i.e. the first level possibly after the dummy root are s-nodes) or it has the same s-node ancestor as the root of the micro tree. In the first case, we determine if it exists in the micro-tree from the *type* and the lookup table. In the second, we store u' by Lemma 3.4.9. If no such u' can be found, we store v's ancestor on the first level of the mini-tree.

For tier-1, we first find u' if it exists. If not, then we have either returned the tier-1 s-node (since it is on the first level of the mini-tree) or we have returned the root of the tree, in which case no tier-1 s-node ancestor exists. If u' exists, we store the tier-1 s-node ancestor u at the tier-2 s-node u' if it exists. We may store this by Lemma 3.4.8.

For descendants, given a node v, we wish to find the next level with tier-1/tier-2 s-nodes. On this level, there is an interval of s-nodes that are descendants of v (which may be empty). We wish to return u_1, u_2 which are the first and last node in this interval which are descendants of v.

First we consider tier-2. For each micro-tree whose last level are s-nodes, and for each v, we store in the lookup table, the range of s-nodes descended from v. For each micro-tree, and for each v, we store whether the external edge is a descendant of v and if so the root of the micro-tree corresponding to the external edge. Finally, we store the range of s-nodes that are descended from the root of the micro-tree. All of these are $O(\lg\lg n)$ bits so we can do this by Lemma 3.4.9. Thus for each v, if it is the root, we are done. Otherwise, we union the ranges of the micro-tree descendants and the possible descendants contributed by the external edge. We note that if there are no such s-nodes because this is the last slab, the range we obtain are either the last level of the entire tree, or a range of tier-1 s-nodes instead.

For tier-1, we first obtain the range of tier-2 s-nodes. If this is indeed the last (tier-2) slab we are done as these nodes are tier-1 s-nodes or no such tier-1 s-nodes can be found. For each tier-2 s-node, we store the range of tier-1 s-node descendants. For each tier-2 s-node we store a bitvector stating whether it has tier-1 s-node descendants. Thus given u_1, u_2 tier-2 s-nodes, we use rank and select to find the next/previous u'_1, u'_2 which have descendants. Then return the first descendant of u'_1 and the last descendant of u'_2. Again the space usage for all the information here is $O(\lg\lg n)$ bits for each tier-2 s-node and so we may store them by Lemma 3.4.11.

Thus we obtain:

Lemma 3.4.12. *Let v be a node. We may report u, u' the closest tier-1/tier-2 s-node ancestor of v. We may also report u'_1, u'_2 the range of tier-2 s-node descendants of v (on the closest tier-2 s-node level below v) and u_1, u_2 the range of tier-1 s-node descendants of v in $o(n)$ bit of space.*

3.4.3 Pre and Level-Order Rank/Select, `prev_internal`, `next_internal`

We now describe how to support operations efficiently in our data structure. We will describe the new operations that we are able to support first - the level-order specific operations, and describe how to support the previous supported operations in Section 3.5.

Preorder Rank In this operation, we are given v by $\tau(v) = \langle \tau_1, \tau_2, \tau_3 \rangle$, and we wish to find the global preorder rank. Let u and u' be the tier-1 resp. tier-2 preorder changers associated with v by Lemma 3.4.8. 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.

Again by Lemma 3.4.8 at each tier-2 preorder changer u', we store the distance to its associated tier-1 preorder changer u which is $O(\lg\lg n)$ bits. At each tier-1 preorder changer, we store its preorder rank which is $O(\lg n)$ bits. In the micro-tree lookup table, we store the distance between v and u'.

Preorder 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 that we have made before 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.

Hence by Lemma 3.4.8, we may store the τ-name of all tier-1 preorder changers, and $\langle \tau_2, \tau_3 \rangle$ of all tier-2 preorder changers. Thus given v we may compute $\tau_1(v)$ and $\tau_2(v)$. Finally we obtain $\tau_3(v)$ using the micro-tree lookup table, found by looking the node at distance $v - u'$ away from u' in the micro-tree.[6]

[6]Using the same techniques, we are also able to support the rank and select operations in `POST`, `IN`, `DFUDS` traversal orders, by applying Lemma 3.4.8.

Level-Order Rank Given a node v by τ-name, we now seek the i with $v = w_i$ (i.e. the level-order rank of v is i since it is the ith node in a level-order traversal). 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 $v = w_i$; In the same way as preorder rank, we store the level-order rank of the tier-1 level-order changer, and the distance between the tier-1 and tier-2 changers at j'. Finally in the lookup table, we store the distance between v and $w_{j'}$.

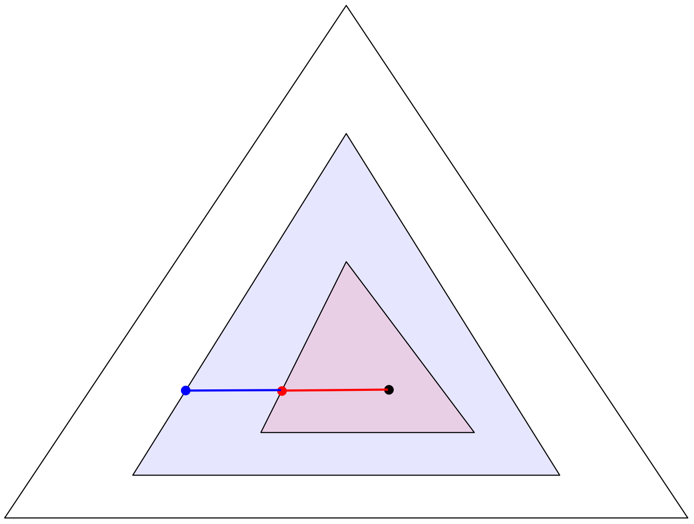


Figure 3.2: A rough sketch of the steps in the computation of level-order rank.

Example 3.4.13. Consider Figure 3.2. Suppose the node given is the black node. The mini-tree is coloured in blue, and the micro-tree in red. The tier-1 level-order changer is the node coloured in blue (i.e w_j), and the tier-2 level-order changer is the node coloured in red (i.e. $w_{j'}$), as these are the node where the level-order traversal changes mini- or micro-trees. We can compute the number of nodes between the red and black nodes in the level-order traversal using the micro-tree lookup table, since the entire segment lies within the micro-tree (this term is $i - j'$). At the red node, we store the distance between it and the blue node (this term is $j' - j$), and at the blue node, we store its level-order rank (the number j). The level order rank of v can be computed by taking the sum of the 3 terms.

Level-Order Select Given the level-order rank i, find $\tau(w_i)$. As in preorder, we find the tier-1/tier-2 changers w_j and $w_{j'}$ associated with w_i. This gives us τ_1 and τ_2. To find τ_3, we again store in the lookup table the node (τ_3) that is $w_i - w_{j'}$ away from $w_{j'}$ in the micro-tree.

`prev_internal` **and** `next_internal` Here we are given $\tau(v)$, and we wish to find `prev_internal`$(v) = \tau(w)$, where w is the last non-leaf node (`degree`$(w) > 0$) preceding v in level order. And `next_internal` is analogous (i.e. the first non-leaf node succeeding v in level-order).

For `prev_internal`, in the micro-tree lookup table, we store whether there is an internal node to the left of v inside the micro-tree between v and its level-order changer u'. For `next_internal`, in the micro-tree lookup table, whether there is an internal node to the right of v between v and the boundary node b immediately preceding the subsequent level-order changer.

For both `prev_internal` and `next_internal`, if the internal node we are looking for is within the micro-tree, we store its τ_3. Otherwise, w does not lie in $\mu^{\tau_1(v)}_{\tau_2(v)}$. By definition, we have `prev_internal`(v) = `prev_internal`(u') and `next_internal`(v) = `next_internal`(b). For `next_internal`, we find the level-order changer b' where `node_rank`$_{\texttt{LEVEL}}(b')$ = `node_rank`$_{\texttt{LEVEL}}(b) + 1$. If b' is a leaf, then `next_internal`$(v) = b'$, otherwise `next_internal`(v) = `next_internal`(b').

At each tier-2 level-order changer, we store $\langle \tau_2, \tau_3 \rangle$ of the previous/next internal node if it is in the same mini-tree. At each tier-1 level-order changer, we store τ of the its previous/next internal node. Thus the node is not in the same mini-tree, proceed to the appropriate tier-1 level-order changer and retrieve the answer there.

3.5 Remaining Tree Queries

In this section, we consider the remaining queries implemented in previous results. See Table 2.1. In many of our operations, the procedure do not need to be changed very much (or at all) from the previous versions of tree covering. Unless otherwise stated, all nodes will be given or returned by their τ-names.

`parent`(v) : If `parent`(v) is in the same micro-tree as v, then we can find this by the micro-tree lookup table. Otherwise, the parent of v in the micro-tree is either the dummy root in which case v is either a tier-1 or tier-2 s-node, and we may store the parent of v explicitly, or v is the root of the micro-tree. If so, then we may store its parent (which must be in the same mini-tree if it is not a s-node). We may do this by Lemma 3.4.9 and Lemma 3.4.11.

`degree` : If v is not the root of the micro-tree, then all of it's children belong to the micro-tree (if v is a promoted s-node, we use the original s-node instead). If v is the root of several micro-trees (but not the root of several mini-trees as well) we store the degree of v in $O(\lg\lg n)$ bits. If it is the root of several mini-trees then we store the degree of v in $O(\lg n)$ bits. We may do this by Lemma 3.4.9.

`child`(v, i) : There is no change needed from previous tree covering solutions, as we will never need to compute children of any dummy roots.

`child_rank`(v) : The only difference is when we find the rank of an s-node. The rank of an s-node is wrong in its mini-/micro-tree because of the dummy root. To do this, in our level-order traversal of the s-nodes Lemma 3.4.11, we store a bitvector of length $O(n/H)$ for tier-1 and $O(n/H')$ for tier-2 which has a 1 for an s-node v is the preceding s-node has a different parent (recall that $H = \lg^3 n$ and $H' = \frac{\lg n}{(\lg\lg n)^2}$). Then `child_rank`$(v)$ is simply the distance between v and the closest preceding 1 in the appropriate (tier-1/tier-2) bit vector.

`depth`(v) : Let u and u' be the closest tier-1 and tier-2 s-node ancestors of v. If u' does not exists, then let u' be the root of the micro-tree containing v. If u' is a dummy root, then u is in the same micro-tree as v. If u does not exist then we are in the first tier-1 slab which contains the root of the entire tree.

From the micro-tree lookup table, we find the distance between v and u'. At each tier-2 s-node and every micro-tree we store the distance to the closest tier-1 s-node from the s-node or the root of the micro-tree (or to the root of the entire tree if no tier-1 s-node ancestors exist). At each tier-1 s-node we store the depth of the s-node. We may do this by Lemma 3.4.9 and Lemma 3.4.11.

The depth of v is the sum of the distances above.

`anc`(v, i) : The solution of [37, §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 3.5.1 (Level ancestors, [11, Thm. 13]). *There is a data structure using $O(n \lg n)$ bits of space that answers* `anc`(v, i) *queries on a tree of n nodes in $O(1)$ time.*

Geary et al.[37] 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 $O(\frac{n}{H}\lg(\frac{n}{H})) = O(n/\lg n)$ 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 total space for jump pointers are

$$\frac{n}{H} \cdot 2\sqrt{H} \cdot \lg n = \frac{n}{\sqrt{H} \cdot \lg n} = \frac{n}{\sqrt{\lg n}}$$

for tier-1 s-nodes and mini-tree roots and

$$\frac{n}{H'} \cdot 2\sqrt{H'} \lg\lg n = n\frac{(\lg\lg n)^3}{\sqrt{\lg n}}$$

for tier-2 s-nodes and micro-tree roots.

Again by Lemma 3.4.11 and Lemma 3.4.9 we may do this in $o(n)$ bits of space.

The query now works as follows (essentially [37, 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 $O(1)$.

`nbdesc`(v), `leaf_size`(v) : Given v, we will compute the number of descendants of v in the micro-tree, the number of descendants of v in the mini-tree but not in the micro-tree, and the number of descendants of v in the entire tree, but not in the mini-tree. `nbdesc`(v) would clearly be the sum of the tree above quantities.

The first is simple, we store this quantity in the micro-tree lookup table. Let u'_1, u'_2 be the range of tier-2 s-node descendants of v obtained from Lemma 3.4.12. The number of descendants of v outside of the micro-tree are descendants of these s-node if they exist and descendants of the micro-tree root corresponding to the external edge. We store the number of descendants of the root of each micro-tree down to the the level of the tier-2 s-nodes. At each tier-2 s-node we store the number of descendants of it in the mini-tree, along with a partial sum of all the tier-2 s-nodes of that level in the mini-tree.

The number of descendants of all tier-2 s-nodes between u'_1 and u'_2 is therefore the difference of the partial sums stored at u'_1 and u'_2.

We repeat the above process for tier-1 s-nodes and mini-tree roots.

For `leaf_size`(v), for the above, we store the number of leaf descendants rather than the total number of descendants.

`height`(v) : 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. We find the range of tier-1 s-node descendants of the root (and also find the difference in depth of these nodes), and using a range maximum data structure the index of the tier-1 s-node with the greatest height. If no such tier-1 s-node descendants can be found then we are on the last tier-1 slab, and the height will be at most $O(\lg H) = O(\lg \lg n)$ and we may store it explicitly.

If v is not a micro-tree root. We again find the range of tier-1 s-nodes descendants. If they do not exist we find the range of tier-2 s-nodes and perform the range maximum on them. For tier-2 s-nodes in the last slab only, we store the height of them explicitly (and a flag such as -1 otherwise). Finally, if no tier-2 s-nodes range exist, then the height of v is the height in the micro-tree or is the height contributed from the external edge (plus the distance to the external edge) which we may lookup.

The range minimum/maximum data structures use space linear in the number of elements which is at most $O(n/H') = o(n)$.

`LCA`(u, v) : The technique of He et al. [44] based on bootstrapping a non-space efficient solution (similar to `anc`) 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 roots added.

`leftmost_leaf`, `rightmost_leaf` : We may use the same technique with very little modifications. Since our mini-/micro-trees are cut not only at external edges but also slabbed edges, we must store at each tier-1 s-node the left/right most leaf. At each tier-2 s-node, the micro-tree containing the left/right most leaf or the tier-1 s-node descendant (via $\langle \tau_2, \tau_3 \rangle$) which contains the left most leaf.

`level_leftmost`, `level_rightmost` : No changes are needed. The previous work showed that the left most nodes of each level listed by level forms a piece-wise constant sequence, and thus can be stored in $o(n)$ bits. The same holds true as we only introduce at most $O(n/H)$ or $O(n/H')$ more blocks in the sequence.

`level_pred`, `level_succ` : This is now subsumed by $\texttt{node\_rank}_{\texttt{LEVEL}}$ and $\texttt{node\_select}_{\texttt{LEVEL}}$. We simply select the node that is 1 more/less than the given node's rank in a level-order traversal.

`leaf_rank`, `leaf_select` : We first consider `leaf_select` in pre-order. The sequence of leafs is a subsequence of all the nodes, and thus in a preorder traversal, the number of pre-order leaf changers (i.e. a leaf v where the previous leaf is in a different mini- or micro-tree) is $O(n/H)$ or $O(n/H')$, and the same technique as $\texttt{node\_select}_{\texttt{PRE}}$ suffices.

For `leaf_rank`, we use the same technique as $\texttt{node\_rank}_{\texttt{PRE}}$, except at every location storing the number of nodes between pre-order changers, we store the number of leafs instead, which is a smaller quantity.

Combining our work in Sections 3.5, and 3.4 we have the main result of this chapter:

Theorem 3.5.2 (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 2.1 in $O(1)$ time.*

3.6 Discussion

In this chapter we used a novel tree decomposition scheme to support level-order operations in succinct trees. This decomposition scheme has the feature that any subtree created either has all their nodes consecutively in a level-order traversal or the number of nodes is much larger (by a factor of $\lg^2 n$ for mini-trees and $(\lg\lg n)^2$ for micro-trees) than the height of the subtree.

By doing so, we are able to store information for *every* level of the subtree which is necessary to support level-order operations. This is because for any of these subtrees, the level-order traversal enters and leaves the subtree on every level, and the information of the traversal needs to be stored.

In this way, we are able to give the first succinct representation of static ordinal trees supporting both level-order operations and depth-first traversal (pre-order, post-order etc.) operations, whereas previously they were separate.

The consequence of this can be seen in subsequent chapters, where we will use this result as a key building block for succinct data structures for many variations of interval graphs.

For future directions, making the work here dynamic is one direction. Dynamic succinct data structures are much rarer than static succinct data structures. However for trees, there exist dynamic succinct ordinal trees (supporting the inserting and deletion of both internal nodes and leaves), supporting all the non-level-order operations in $O(\lg n/\lg\lg n)$ time [67]. It is a very nice avenue of work to generalize the ideas in this chapter to the dynamic setting.

Next avenue of work is succinct forests. That is we have a collect of trees $T_1, \ldots T_k$. and we wish to support traversals on them. A preorder traversal is trivial, as we may add a dummy root with each of the trees as children of that dummy root. However, in this manner, the level-order traversal is incorrect as the traversal would repeated enter and exit each of the trees. We would like to represent the trees so that the level-order traversal traverse each of the trees sequentially.

A sketch of how this can be done is the following: Add a dummy root as above and slab. For each slab, add a dummy for each tree, then a dummy root overall, then decompose. In this manner, we may show that the number of changers is not too large (any changer for tree in the i-th slab can be charged to $O(H)$ nodes of that tree in the previous slab), and careful work would be needed for the first slab, where we cannot charge the space to a previous slab. Details on supporting all the operations would also need to be worked out as well, but are likely not too different from what is presented in this chapter.

There is work in reducing the redundancy (i.e. the $o(n)$ term) for succinct trees. Normally, this term is on the order of $O(n\lg\lg n/\lg n)$, but there is work which reduces this term to $O(n/\lg n)$ [69]. In this chapter, we have not explicitly analyzed this term, but

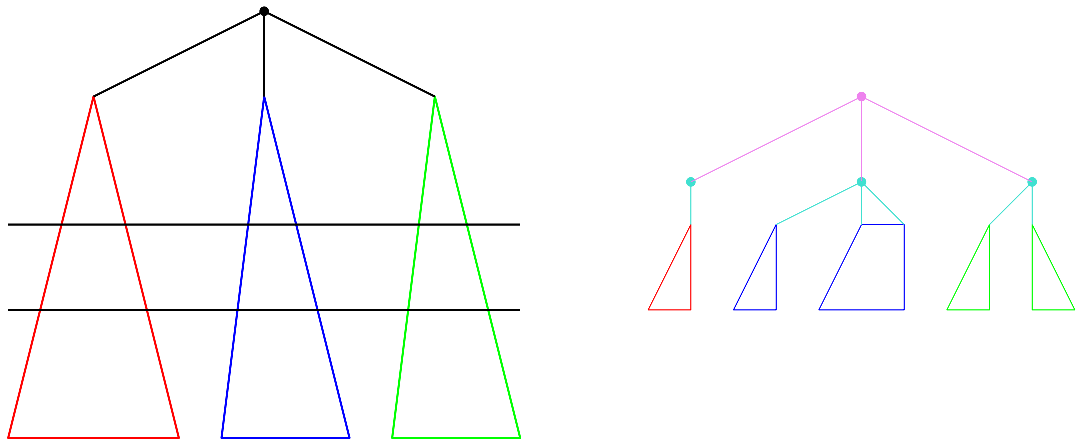


Figure 3.3: We convert the forest into a tree by adding a dummy root. Then for each slab which consists of a forest, instead of adding a single dummy root to convert it into a tree, we add 2 layers of dummy roots. First layer to join together all trees that belong to the same original tree, then add the dummy root to join all the trees together.

it is on the order of $O(n(\lg\lg n)^c/\lg n)$ for some constant $c \geq 3$, since we store $\lg\lg n$ bits at each tier-2 changer, and there are $n/H' = O(n(\lg\lg n)^2/\lg n)$ such changers.

Finally, we have not considered all the operations that can be done in level-order, just those that are needed for the support of our interval graph data structures. However, other level-order operations should be easily implementable using the framework developed here.

Chapter 4

Interval Graphs

In this and the next two chapters we will study data structure problems for interval graphs (Definition 2.1.10). Specifically, in this chapter, we will study static interval graphs and related classes of graphs. The chapter is organized as follows: we begin with a brief introduction in Section 4.1, followed by a review of relevant previous works in Section 4.2, and an overview of existing results that we will be using in Section 4.3. We will study the distance oracle in interval graphs in Section 4.4, study bounded degree and bounded chromatic number interval graphs in Section 4.5, prove lower bounds related to the graphs studied in Section 4.6. Finally, we will summarize our results and discuss open problems in Section 4.7.

4.1 Introduction

Graphs are one the most widely used types of data. In this chapter, we study navigation oracles (i.e. data structures that allow efficient traversal through a graph, through the operations of `adjacent`, `degree` and `neighborhood`) and *succinct distance oracles*, i.e., data structures that efficiently compute the length of a shortest path between two vertices, for interval graphs and related classes of graphs.

To support distance queries, researchers have considered the problem of designing data structures called *shortest path oracles* or *distance oracles*. These data structures are constructed by preprocessing the given graph G, such that, given a pair of query vertices x and y of G, the *shortest path query*, which asks for the list of vertices on the shortest path between x and y, or the *distance query*, which asks for the distance between x and y, can be answered efficiently. A naive solution is to precompute information between all pairs of vertices, and this can answer a distance query in constant time but uses $\Theta(n^2)$ space, where n is the number of vertices in the graph. As quadratic space is believed to be necessary for fast distance queries, to improve space efficiency, much work has been done to design approximate distance oracles for both weighted [80, 21, 78] and unweighted graphs [70, 1]. For example, given a pair of vertices x and y at distance d, the $\Theta(n^{5/3})$-word distance oracles of Pătraşcu and Roditty [70] can compute in constant time an approximate distance in $[d, 2d+1]$. It can also return a path between x and y with length upper bounded by $2d+1$ in $O(d)$ time. This result was generalized by Abraham and Gavoille to design an $\tilde{\Theta}(n^{1+2/(2k-1)})$-word[1] oracle that can compute in $O(k)$ time a distance in $[d, (2k-2)d+1]$, for any integer $k > 1$.

These distance oracles often use $\Theta(n^{1+\Omega(1)})$ space to provide fast support for distance queries. However, modern applications often process large graphs, and data structures with such space costs tend not to fit in main memory. Therefore, a trend in the design of distance oracles is to take advantage of the structural properties of various classes of graphs to design more space-efficient solutions. The classes of graphs considered include both sparse graphs such as planar graphs [55, 54] and chordal graphs [76, 65].

Interval graphs are the intersection graphs of intervals on the real line and have applications in operations research [9] and bioinformatics [84]. Distance oracles are widely studied; for an overview of the extensive literature see [78, 85, 80, 70].

4.2 Previous Work

In this section, we give a wide variety of previous works on succinct graph data structures, and in subsequent chapters focus on previous works related to the topics of those chapters.

Several succinct representations of (subclasses of) graphs have been studied, e.g., for general graphs [30], k-page graphs [49], certain classes of planar graphs [24, 23, 17], separable graphs [14], posets [59], distributive lattices [64], chordal graphs [65], path graphs [8], series-parallel, block-cactus, 3-leaf-power graphs [19], permutation graphs [81]. Acan et al. [2] showed how to represent an *interval graph* on n vertices in $n \lg n + (3+\epsilon)n + o(n)$ bits to support `degree` and `adjacent` in $O(1)$ time, `neighborhood`(v) in $O($`degree`$(v))$ time and `spath`(u, v) in $O(|$`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

[1]The $\tilde{O}$ and $\tilde{\Theta}$ notation hides `polylog`(n) multiplicative factors.

proper interval graph and a *k-proper/k-improper interval graph* (defined below, Definition 4.3.2, Definition 4.3.3) in $2n + o(n)$ and $2n \lg k + 6n + o(n \lg k)$ bits, respectively, supporting the same queries.

For bounded degree interval graphs and bounded chromatic number interval graphs, Chakraborty and Jo [18] showed lower bounds of $\frac{1}{6} n \lg \sigma$ bits where σ is the maximum degree in bounded degree interval graphs and the chromatic number in bounded chromatic number interval graphs. For an upper bound, in the case of bounded degree interval graphs they gave a data structure occupying $n \lg \sigma + O(n)$ bits of space, supporting `degree` and `adjacent` in $O(1)$ time, `neighborhood`(v) in $O($`degree`$(v))$ time and `spath`(u, v) in $O(|$`spath`$(u, v)|)$ time. For bounded chromatic number interval graphs, they gave a data structure occupying $(\sigma - 1)n + O(n)$ bits of space supporting `adjacent` in $O(\lg \lg \sigma)$ time, `degree` and `neighborhood` in $O(\sigma \lg \lg \sigma)$ time.

4.3 Preliminaries

As a bit-vector is the most fundamental data structure in succinct data structures, we will need to make use of it. We will also require the generalization to strings.

Lemma 2.3.2 ([62]). *A bit-vector of length n can be represented in $n + o(n)$ bits to support* `access`, `rank`, `select` *in $O(1)$ time.*

Lemma 2.3.6 ([39]). *A string S of length n over an alphabet of size σ, can be succinctly represented using $n \lg \sigma + o(n \lg \sigma)$ bits to support* `rank`, `access` *in $O(\lg \lg \sigma)$ time and* `select` *in $O(1)$ time.*

The main building block we will use for the `distance` query will be succinct trees presented in Chapter 3.

Theorem 3.5.2 (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 2.1 in $O(1)$ time.*

We will also make use of the data structures for orthogonal range operations.

Lemma 2.5.6. *Given n 2-dimensional points, and let f be the time cost of* `rank`, *g be the time cost of* `decode`. *Then we are able to support 3-sided reporting queries (of the form $[x_1, x_2] \times [-\infty, y]$, to support the symmetric cases, we duplicate the structure) in time $O(f + k \cdot g)$ where k is the number of points reported. The space cost is $2n + o(n)$ plus the space needed to support the* `rank` *and* `decode` *operations.*

4.3.1 Interval Graphs

We will now expand on Definition 2.1.10. In Definition 2.1.10, we used a set of nodes that is a path in a tree that was itself a path. Equivalently, if we number the nodes of the path by strictly increasing real numbers, then the set of nodes can be written as an interval on the real line. By doing so, we have the equivalent definition of interval graphs:

Definition 4.3.1. A graph G is an ***interval graph*** if it is the intersection graph of intervals on the real line. That is for every vertex v, s_v is a set containing the real numbers in the interval $[l_v, r_v]$. In this way, we will name the interval $I_v = [l_v, r_v]$.

For an interval graph, we may sort the end points so that they lie in the range $[1 \dots 2n]$ while preserving the intersection structure (if the left endpoint of an interval is the same as the right end point of a different interval, the left end point comes first to preserve intersections). See Figure 4.1.

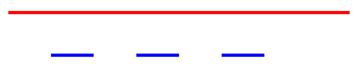
For each vertex v, we will treat v as a number, which is the index of its left endpoint among all left end points, therefore, the statement $u < v$, which compares the numbers assigned to the vertices will also implicitly compare the left end points of the two vertices.

Some subclasses and related classes of interval graphs that we will study are:

Definition 4.3.2. A ***proper interval graph*** is an interval graph where there exists a way to associate each vertex v with an interval $I_v = [l_v, r_v]$ such that no two interval nest. That is for all u, v, $I_v \cap I_u \notin \{I_v, I_u\}$.

Definition 4.3.3. An interval graph is ***k-proper*** if there exists an way to associate each vertex v with an interval such that for every vertex v, the number of intervals containing it is at most k. An interval graph is ***k-improper*** if there exists an way to associate each vertex v with an interval such that for every vertex v, the number of intervals it contains is at most k.

Example 4.3.4. Consider the graph defined by the following intervals.



In this way of representing the graph via intervals, the red interval contains the 3 blue intervals, and each blue interval is contained by the red interval. Thus this graph is 1-proper and 3-improper. We note that since the definition is an existential statement, this graph could also be 2-improper or even 1-improper, if a better representation can be found.

Definition 4.3.5. An interval graph is a ***bounded degree interval graph*** with respect to a parameter σ if the maximum degree over all vertices is σ. An interval graph is a ***bounded chromatic number interval graph*** with respect to a parameter σ if we may assigned a colour $c_v \in [1, \ldots, \sigma]$ to each vertex such that any two adjacent vertices have different colours.

Definition 4.3.6. A graph is a ***circular arc graph*** if it is the intersection graph of arcs on a circle.

We will first briefly describe the data structure of [2] as we will build our data structure on top of it, and in the case of bounded degree and bounded chromatic number interval graphs, we will need to interface with it closely.

We store a bit vector B of length $2n$ with index corresponding to the beginning of an interval as a 0 and the index corresponding to the end of an interval as a 1. We name the intervals 1 to n by left endpoint so that the i-th interval's left endpoint is the index of the i-th 0. Thus the left endpoint of any interval can be obtained using the `select` operation. To obtain the right endpoint, there are 3 obvious ways to do it: store the right endpoint of the i-th interval explicitly; for each interval $[l_v, r_v]$, store the difference $r_v - l_v$; or store the rank of the right endpoint (the i-th 0 corresponds to the j-th 1, defining a permutation). In all of these methods, it takes $O(1)$ time to compute the right endpoint of any interval and uses $n \lg n$ bits of space in the worst case.

For `degree`(v), we count the number of intervals that do not intersect the given interval $[l_v, r_v]$. That is all intervals of the form $[x, y]$ with $y < l$ or $x > r$. The count of the first type we obtain by counting the number of 1s before the index l in the bitvector with rank. The second we count the number of 0s after the index r in the bitvector.

Finally for `neighborhood`(v), there are two types of neighbours. The first are neighbours ($[x, y]$) where $l_v < x < r_v$. To output these, we simply output each interval corresponding to a 0 in the indices between l_v, r_v using `rank` and `select`. The second type are those with $x < l_v$. For these we must have $y > l_v$. If we treat the intervals as points in 2D space, then these are exactly the points within the 3-sided rectangle $[0, l_v) \times (l_v, \infty]$. Given the left end point of an interval, `rank` is simply $\texttt{rank}_B$. As we have a way to compute the right endpoints given left endpoints (i.e. `decode`), we may solve this using $2n$-bits in $O(1)$ time. We note that in their paper, the technique they used is exactly the method to solve 2-dimensional 3-side reporting queries, though they did not explicitly say it.

As their solution's bottleneck for space is the data structure computing the right endpoints, we will leave this part as a placeholder. Thus to use their data structure, it suffices to support the operation `decode`: given the rank of any interval i, we need to be able to compute both the left and right endpoints. Once we have this, we may store the bitvector to compute degree and a 2D range reporting data structure for neighbourhood.

We may then restate the theorem of Acan et al. as:

Theorem 4.3.7 ([2]). *Let G be an interval graph. Let D be a data structure that can compute the right endpoints of the i-th interval using $g(n)$ bits of space and $f(n)$ time. Then we may support* `adjacent`, `degree` *in $O(f(n))$ time,* `neighborhood` *in $O(f(n))$ time per neighbour. This takes $g(n) + 4n$ bits of space. Moreover for any vertex v we may retrieve the interval $[l_v, r_v]$ in $O(f(n))$ time.*

Furthermore there exists a data structure D where $f(n) = O(1)$ and $g(n) = n \lg n$.

Lastly to handle the case of disconnected interval graphs, we need to be able to know if two vertices are in the same component. To do this, we will use the equivalence class data structure of El-Zein et al.

Lemma 4.3.8 (Thm. 7 of [28]). *Given a partition of an n-element set into equivalence classes, $O(\sqrt{n})$ bits are necessary and sufficient for storing the partition and to answer the equivalence query in constant time if each element is to be given a unique label in the range $\{1, 2, \ldots, n\}$.* [2] *Furthermore, the labelling is given by sorting the equivalence classes from smallest to largest, then labelling the elements in order.*

4.4 Distances in Interval Graphs

In this section, we will describe how to support the `distance` query in interval graphs by augmenting the data structure of Acan et al. [2] using $O(n)$ additional bits.

As interval graphs are a subclass of chordal graphs, we will be using the algorithm of Munro and Wu [65][3] 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. We define $\texttt{parent}(v) = \min B_v$ (i.e interval in B_v with the smallest left endpoint).

The shortest path algorithm given in [65] when reduced to interval graphs is similar to the one in [2]. Given $u < v$, we compute the shortest path by checking if u and v are

[2]The input to the equivalence class query are the labels of the two elements.

[3]This algorithm is not part of this thesis, but rather can be found in the cited paper or the author's MMath thesis.

adjacent. If so, add u to the path; otherwise, add $\texttt{parent}(v)$ to the path and recursively find $\texttt{spath}(\texttt{parent}(v), u)$.

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 (denoted as the ***distance tree***) as follows: for every vertex $v = 1, \ldots, n$ (in that order), add node v to the tree as the rightmost (last) child of $\texttt{parent}(v)$; see Figure 4.1 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 4.4.1 below.

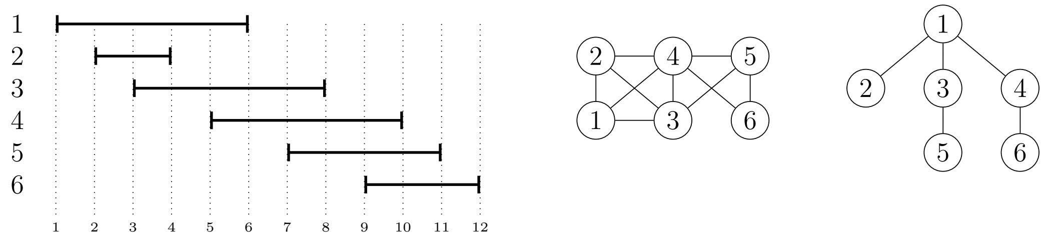


Figure 4.1: An Interval Graph (middle) with Interval Representation (left), and distance tree constructed (right).

We note that the above construction is undefined for a disconnected graph, as the leftmost interval of a component would have an undefined parent (or rather the parent is itself, but that would not define a tree). 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). To avoid reporting this edge in queries, we use Theorem 4.3.8 to detect the boundaries of components (by check if $v - 1$ and v are in the same equivalence class). Thus we may detect if v is the first or last vertex in its component. By Theorem 4.3.8, we first order the components by their sizes. Any distance queries (between u and v) will first check if the two vertices are in the same component. 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.

This uses an additional $O(\sqrt{n}) = o(n)$ bits of space, which we will ignore as it is a lower order term.

Lemma 4.4.1 (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.

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 $s_u = \min B_u$. $\square$

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. See Figure 4.2.

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 $\texttt{spath}$ 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 $\texttt{spath}$ algorithm.

We may summarize the above as the following:

Lemma 4.4.2. *Let G be an interval graph with distance tree T. Let u, v be two vertices of G with $\texttt{node\_rank}_{\texttt{LEVEL}}(u) > \texttt{node\_rank}_{\texttt{LEVEL}}(v)$. Consider the node to root path of u as $u = u_1, \ldots, u_k = r$. Let i be the first index where $l_{u_i} \leq r_v$. Then a shortest path from u to v is $u = u_1, \ldots, u_i, v$.*

Furthermore, $\texttt{depth}(u_i)$ is either $\texttt{depth}(v) - 1$ or $\texttt{depth}(v)$ or $\texttt{depth}(v) + 1$.

The extra space needed is to store the tree T, using $2n + o(n)$ bits. The results described above are summarized in the following theorem:

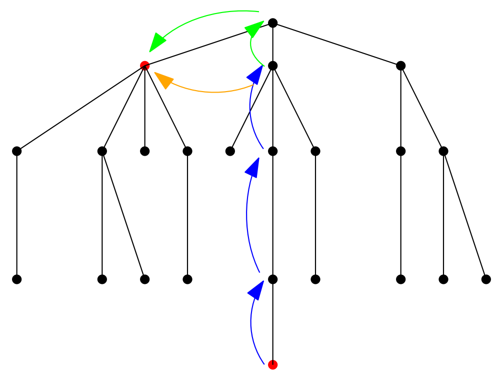


Figure 4.2: The computation on a shortest path on the distance tree between the two red nodes (u at depth 2 and v at depth 5). We repeatedly apply the `parent` operation. The orange arrow is taken if those two nodes are adjacent. If they are not, the green path is taken instead. Due to this, the penultimate vertex on the path cannot have depth smaller than $\texttt{depth}(u)-1$ nor greater than $\texttt{depth}(u)+1$. To compute the distance, we check which of the 3 possible nodes is the penultimate one, using `anc`.

Theorem 4.4.3 (Succinct interval graphs with distance)**.** *An interval graph G can be represented using $n \lg n+(5+\epsilon)n+o(n)$ bits to support* `adjacent`*,* `degree` *and* `distance` *in $O(1)$ time,* `neighborhood` *in $O(\texttt{degree}(v)+1)$ time, and* `spath`(u,v) *in $O(\texttt{distance}(u,v)+1)$ time.*

Applying this to the more abstract data structure discussed in Theorem 4.3.7, we obtain the following more abstract version of the data structure with distances:

Theorem 4.4.4. *Let G be an interval graph. Let D be a data structure that can compute the right endpoints of the i-th interval using $g(n)$ bits of space and $f(n)$ time. Then we may support* `adjacent`, `degree`, `distance` *in $O(f(n))$ time,* `neighborhood` *in $O(f(n))$ time per neighbour and* `spath` *in $O(f(n)+\texttt{distance})$ time. This take $g(n)+6n$ bits of space. Moreover for any vertex v we may retrieve the interval $[l_v, r_v]$ in $O(f(n))$ time.*

Furthermore there exists a data structure D where $f(n)=O(1)$ and $g(n)=n\lg n$.

Finally we note that this augmentation can without changes be applied to subclasses of interval graphs; we thus obtain the following theorem (where the increase in space over the data structure of Acan et al. [2] is $2n$ bits, and the time complexities are the same):

Theorem 4.4.5 (Succinct k-proper/-improper interval graphs with distance)**.**

A k-proper (k-improper) interval graph[4] *G can be represented using $2n \lg k + 8n + o(n \lg k)$ bits to support* `degree`*,* `adjacent`*,* `distance` *in $O(\lg\lg k)$ time,* `neighborhood` *in $O(\lg\lg k \cdot$ (*`degree`*$(v) + 1))$ time and* `spath`*(u, v) in $O(\lg\lg k \cdot$ (*`distance`*$(u, v) + 1))$ time.*

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 is presented there. In this case of proper interval graphs, simply adding our distance tree is not good enough.

4.4.1 Proper Interval Graphs

Recall that a proper interval graph is an interval graph that admits an interval representation with no interval 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 $2n - O(\lg n)$ bits [42, Thm. 12].

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 proper interval graphs, the representation is succinct. First, the neighbourhood of a vertex can be succinctly described:

Lemma 4.4.6. *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 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: $\ell_{u_1} < \ell_w \leq \ell_v < r_{u_1} < r_w$. Thus I_v intersects I_w and v is adjacent to w. So the neighbourhood 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` [5].

[4] We note that Klavík et al. [51] consider a closely related class of interval graphs, k-NestedINT that is similar to (and contains) the class of $(k-1)$-improper interval graphs considered by Acan et al. [2], 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'.

[5] We will of course need to perform the same components check as before as well.

`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 $\texttt{parent}(v)$; then u and v are adjacent if and only if $\texttt{parent}(v) \leq u$. Correctness follows from the fact that the neighbourhood of v is a contiguous interval.

`neighborhood`: Let the neighbourhood of v be $[u_1, u_2]$. By the definition of $\texttt{parent}(v)$, we have that $u_1 = \texttt{parent}(v)$ (unless v is leftmost in its component; 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 4.4.7. *If v is a leaf, then $u_2 = \texttt{last\_child}(\texttt{prev\_internal}(v))$; otherwise we have $u_2 = \texttt{last\_child}(v)$.*

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 $w = \texttt{last\_child}(\texttt{prev\_internal}(v))$ denote this node. By definition, $\texttt{parent}(w) < v$ and $w \geq v$. As the neighbourhood 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 $\texttt{parent}(w + 1) > v$. Thus by the previous argument, it cannot be adjacent to v. □

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

The results in this section are summarized in the following theorem; we note that the succinct representation of neighbours allows to report those faster than what was presented in Acan et al.'s representation, which returned each of the neighbours one at a time, using $O(\texttt{degree}(v))$ time.

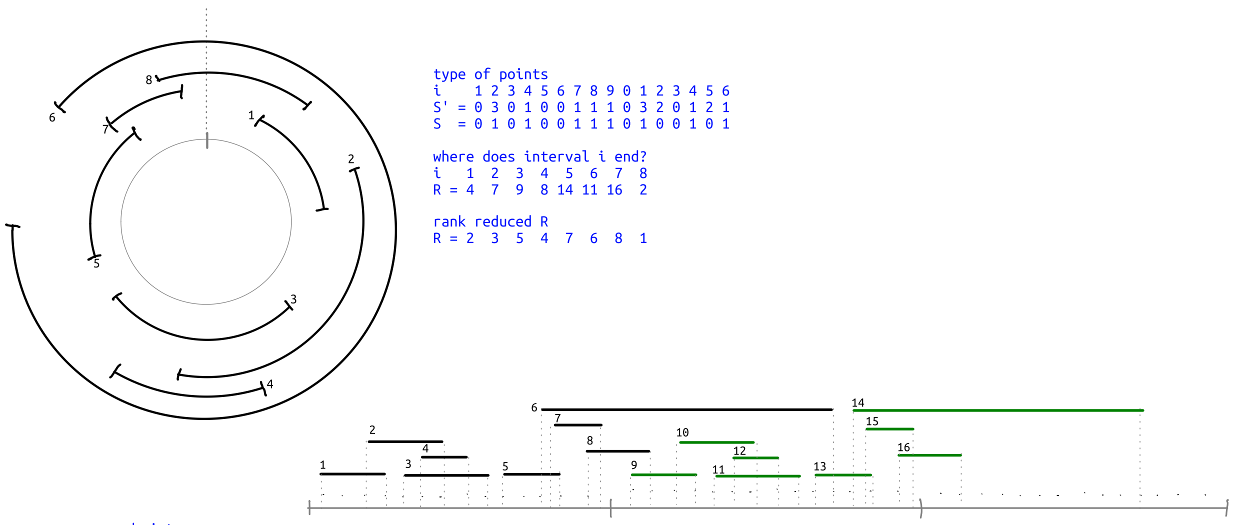
Theorem 4.4.8 (Succinct proper interval graphs with distance)**.** *An interval graph can be represented in asymptotically optimal $2n + o(n)$ bits while supporting* `adjacent`*,* `degree`*,* `neighborhood` *and* `distance` *in $O(1)$ time, and* $\texttt{spath}(u, v)$ *in $O(\texttt{distance}(u, v))$ time.*

4.4.2 Circular Arc Graphs

We finally show how to extend our distance oracles to circular-arc graphs. We follow the notation of [2] for circular-arc graphs, in particular, we assume that we are given left and right endpoints of the vertices' arcs in $[l_v, r_v] \in [2n]$ for $v = 1, \ldots, n$, all endpoints are distinct, and $l_1 < \cdots < l_n$, i.e., vertex ids are by sorted left endpoints. Moreover, v is a *normal* vertex if $l_v < r_v$; otherwise it is a *reversed* vertex corresponding to the arc $[l_v, 2n] \cup [1, r_v]$. We assume that G is connected; if not, G is actually an interval graph, and we can use Theorem 4.4.3.

Acan et al. [2] describe two succinct data structures for circular-arc graphs: one based on succinct point grids (the "grid version") that supports all operations of Theorem 4.3.7, but each with a $\Theta(\lg n / \lg\lg n)$-factor overhead in running time (see [2, Thm. 5]), 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. We describe how to augment either of these to also answer `distance` queries (in $O(\lg n / \lg\lg n)$ 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 4.3 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 4.3: 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 [35] have shown that this construction preserves distances in the following sense:

Lemma 4.4.9 ([35, Lem. 6]). *Let $G = ([n], E)$ be a circular-arc graph with arcs $[l_v, r_v]$ where endpoints are distinct and in $[2n]$ and $l_1 < \cdots < l_n$. Define $\tilde{G} = ([2n], \tilde{E})$ as the interval graph with the following sets of intervals: for every normal vertex v, include $[l_v, r_v]$ and $[l_v+2n, r_v+2n]$ and for every reversed vertex u, include $[r_u, l_u+2n]$ and $[r_u+2n, l_u+4n]$. Then for any $u < v$, we have (identifying vertices with the ranks of their left endpoints)*

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

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., $O(\lg n/\lg\lg n)$ 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 `R` in Figure 4.3.) 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 3.5.2 in $4n + o(n)$ bits, as well as the auxiliary data structures of Acan et al. (without r) from Theorem 4.3.7, all of which occupy $O(n)$ bits. Together this shows the following result.

Theorem 4.4.10. *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* $O(\lg n/\lg\lg n)$ *time,*
`neighborhood`(v) *in* $O((\mathtt{degree}(v)+1)\cdot\lg n/\lg\lg n)$, *and*
`spath`(u, v) *in* $O((\mathtt{distance}(u,v)+1)\cdot\lg n/\lg\lg n)$ *time; or*

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

4.5 Bounded Degree and Bounded Chromatic Number Interval Graphs

In this section, we consider a different subclass of interval graphs by introducing a parameter. The two parameters we are interested in is the maximum degree of any vertex, and the chromatic number of the graph. The chromatic number of a graph is the minimum number of colours needed so that every vertex is assigned one colour, and two adjacent vertices are assigned difference colours.

We will denote σ as the maximum degree or chromatic number of an interval graph, and our goal is to store interval graphs with maximum degree at most σ and interval graphs with chromatic number at most σ in $n \lg \sigma + o(n \lg \sigma)$ bits. As we will see later, the space bound we are aiming for is optimal and these are succinct data structure for $\sigma = \omega(1)$.

For both of these cases, we will use the abstract version of the interval graph data structure Theorem 4.4.4, where the only data structure we need to construct is D, where given a vertex v (which by our naming scheme, is a number indicating the index of the left end point of its interval among all left end points of intervals), computing the right end point of its interval.

In the case bounded degree interval graphs, we have the following result of Chakraborty and Jo [19].

Theorem 4.5.1. *Let G be a interval graph with maximum degree σ, then there exists a data structure D computing the right end points of intervals in $O(1)$ time and $n \lg \sigma + O(n)$ bits of space.*

The proof is quite simple. We note that in our discussion of Theorem 4.3.7, there are many ways to compute the right endpoints of intervals, three of which are: writing down the endpoint explicitly, writing down the difference $r_v - l_v$ and writing down a permutation. In the case of bounded degree interval graphs, writing down the difference is the most natural.

The key observation is that for an interval $[l_v, r_v]$, it must have at least $\frac{r_v - l_v}{2}$ neighbours. This is because every end point $l_v < p < r_v$ belongs to a neighbour, and every such neighbour contributes at most two (one for each of its endpoints) points in this interval. Thus if the maximum degree is σ, then for every vertex v, $r_v - l_v = \Theta(\sigma)$. Hence storing the difference for every vertex uses $n \lg \sigma + O(n)$ bits of space and can compute the right endpoint in $O(1)$ time.

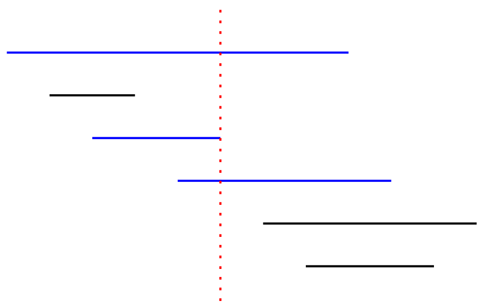
Now we consider the case for bounded chromatic number interval graphs. We will give two data structures, first a succinct data structure using $n \lg \sigma + o(n \lg \sigma)$ bits of space but with query times $O(\sigma \lg n)$. The time complexity is quite bad but the goal is the give a matching upper bounded to the lower bound we will prove. The second is a compact data structure using $2n \lg \sigma + o(n \lg \sigma)$ bits with query times $O(\lg \lg \sigma)$.

4.5.1 Bounded Chromatic Number Interval Graphs

We will construct the data structure D to compute the right end points, using $n \lg \sigma$ bits of space.

Consider the intervals of the interval graph. For each index corresponding to the end of an interval, it corresponds to a previous start of an interval. As the maximal cliques are of size at most σ,[6] the number of unclosed intervals at any point is at most σ (as otherwise, all of the unclosed intervals are adjacent and would form a clique), thus we may store the index of the left end point of the unclosed interval matching each index corresponding a right end point of some interval in $\lg \sigma$ bits. Thus the total space required is $n \lg \sigma + O(n)$ bits.

Example 4.5.2. In the figure below, consider the right endpoint index corresponding to the dotted red line.



At this point, the number of unclosed intervals is 3 outlined by the blue intervals. We store a number between 1 and 3, stating which interval (ordered by left endpoint) this right endpoint closes. In this case, since the interval being closed is the second smallest right endpoint, we store a 2. For each of the 6 right endpoints (in sorted order) we similarly store a number. In this example, the 6 numbers would be 2,2,1,1,2,1.

[6]Interval graphs are perfect graphs, whose maximum clique size is equal to the chromatic number, this can be seen by a straightforward greedy colouring algorithm.

To show case the algorithm below, consider the third interval by left endpoint (i.e. the interval being closed at the red dotted line). At its left endpoint, there are 3 unclosed intervals, so its rank is 3. As we sweep to the right, we encounter the first right endpoint, which has number 2, so we reduce the rank by 1 (as one of the smaller intervals is now closed). The next right endpoint also has number 2. Since it matches our current rank, we have found our matching right endpoint.

To support the navigational operations we will need to be able to find the index of the right endpoint of the interval given the left end point of an interval.

To do this we use the following algorithm: at the left endpoint, the rank of the vertex v is the excess (the number of unclosed intervals). For each following index that corresponds to the right endpoint of an interval there are 3 cases:

- It closes an interval with rank larger than the current rank: there is nothing to be done and we skip it.
- It closes an interval with rank equal to the current rank: we have found our closing parenthesis.
- It closes an interval with rank less than the current rank: we decrease our current rank by 1.

In this way, we are able to compute index of the end of the interval in $O(n)$ time. To decrease this, we store the indices (left endpoints) of the current (un-closed) intervals at every $\sigma \lg n$ right endpoints of intervals - and denote these as shortcuts. We binary search these to find the last shortcut where our query interval is still open - so our matching right endpoint is between this shortcut and the next. We then apply the above linear search algorithm. This takes $O(\sigma \lg n)$ time instead, and uses $O(n)$ extra bits. The result can be summarized as:

Theorem 4.5.3. *Let G be a interval graph with chromatic number σ. Then we may represent G using $n \lg \sigma + O(n)$ bit of space and supports* `adjacent`, `degree`, `neighborhood`, `distance` *in $O(\sigma \lg n)$ time (or per neighbour) and* `spath` *in $O(\sigma \lg n + d)$ time.*

We now focus on reducing the time complexity of the operations, at the cost of slightly increasing the space.

We consider the bitvector denoting pattern of where the intervals begin and end for the graph G. Colour the graph using at most σ colours. We create the string S by replacing each 0 and 1 by the colour of the vertex that is denoted by the bit. For each colour, we note that any odd occurrence of the colour in the string must be the start of the interval of a vertex of that colour, and the following even occurrence of that colour is the matching end of the interval.

We store this string S using 2.3.6. As the length of the string S is $2n$, the space required is $2n \lg \sigma + o(n \lg \sigma)$ bits.

For the right endpoint, we find the left endpoint l_v of the vertex v. We then find the colour c of the vertex using `access`(l_v) on S and then find the matching right endpoint using `rank` and `select` on that colour. (`select`$(c,$ `rank`$(c, l_v) + 1)$).

Applying Theorem 4.4.4, we obtain that `neighborhood` uses $O(\lg \lg \sigma)$ time per neighbour. However, we note that every interval that intersects $[l, r]$ of the type $x < l$ (which necessitates the 2D range search) are all adjacent to each other, thus all must have different colours. Thus the number of vertices of this type is at most σ. Thus the run time is actually at most $O(\sigma \lg \lg \sigma + d)$ as our range search returns at most σ points. The compact data structure can be summarized as:

Theorem 4.5.4. *Let G be a interval graph with chromatic number σ. Then we may represent G using $2n \lg \sigma + o(n \lg \sigma)$ bit of space and support* `adjacent`, `degree`, `distance` *in $O(\lg \lg \sigma)$ time,* `neighborhood` *in time $O(\min(d \lg \lg \sigma, \sigma \lg \lg \sigma + d))$ time and* `spath` *in $O(\lg \lg \sigma + d)$ time.*

4.6 Lower Bounds

In this section, we derive lower bounds of $n \lg \sigma - o(n \lg \sigma)$ for both bounded degree interval graphs and bounded chromatic number interval graphs. Improving the result of Chakraborty and Jo [19] of $\frac{1}{6} n \lg \sigma$.

Given an interval graph G, consider the set of maximal cliques $C_1, \ldots C_k$ of G with $k \leq n$. For any vertex v of G, let $S_v = \{C_i; v \in C_i\}$ be the set of maximal cliques that contain v. Booth and Leuker and Fulkerson [15, 34] showed that G is an interval graph if there exists an ordering of the maximal cliques such that for each vertex v, S_v consists of consecutive maximal cliques. Thus we will now assume that our ordering of maximal cliques satisfies this property (consecutive clique property).

In light of this fact, we will now write $S_v = [l_v, r_v]$ to denote that $S_v = \{C_{l_v}, C_{l_v+1}, \ldots, C_{r_v}\}$. We say that a vertex v with support `support`$(v) = [l_v, r_v]$ begins in clique l_v and ends in clique r_v. Denote the support of a vertex $\texttt{support}(v) = S_v = [l_v, r_v]$. Let $V_1 \subset V$ be a subset of the vertices, we will extended the definition of support naturally by $\texttt{support}(V_1) = \cup_{v \in V_1} \texttt{support}(v)$.

To show the lower bound, we will demonstrate a class of interval graphs by an ordering of maximal cliques. Furthermore, each interval graph in this class will have at most two ordering of maximal cliques which satisfy the consecutive clique property, thus showing that we never construct each graph more than twice.

For vertices u, v, we say that they *overlap* if $S_v \cap S_u \notin \{S_v, S_u, \emptyset\}$. That is they overlap if they intersect but one does not contain the other. For an interval graph G, denote the overlap graph $O(G)$ on the same vertex set but rather than using intersect of the supports as the adjacency criteria, we use overlapping as the criteria. Thus $(u, v) \in E(O(G))$ if u, v overlap. As u, v overlap imply that they intersect, $O(G)$ is a subgraph of G. In particular, any connected component of $O(G)$ is connected in G.

The theorem of [53] gives a condition for the number of orderings of maximal cliques of G to be constant.

Theorem 4.6.1. *If $O(G)$ has a single component, then the number of orderings of the maximal cliques of G satisfying the consecutive clique property is at most 2 (one ordering and the reverse).*

Next we give some structural properties of the overlap components of an interval graph.

Lemma 4.6.2. *Let V_1 be a connected component of the overlap graph of an interval graph G, then $\texttt{support}(V_1)$ is a single interval $[l, r]$.*

Proof. Suppose for a contradiction that there are at least two intervals $[l_1, r_1], [l_2, r_2]$ that do not intersect. Then any vertex v with $\texttt{support}(v) \subset [l_1, r_1]$ and any vertex u with $\texttt{support}(u) \subset [l_2, r_2]$ cannot be adjacent to each other, so the two sets $\{v; \texttt{support}(v) \subseteq [l_1, r_1]\}$ and $\{u; \texttt{support}(u) \subseteq [l_2, r_2]$ are disconnected. But by assumption, V_1 is overlap connected, and thus connected, contradiction. $\square$

Lemma 4.6.3. *Let V_1 with support $[l, r]$ such that $l \neq r$ be a connected component of the overlap graph of an interval graph. Then for any value $x \in [l, r]$, there exists a vertex $v \in V_1$ such that $l_v \leq x \leq r_v$ with $l_v \neq r_v$. Furthermore, if $x \neq l$, then we may choose $l_v < x \leq r_v$.*

Proof. First we note that no vertex can have $l_v = r_v$ since it does not overlap any other vertex.

In the case that $x = l$ then as the support of V_1 is $[l, r]$ there exists a vertex v with $l_v = l$ and $r_v > l$.

Now suppose that $x \neq l$ and suppose that no such vertex can be found. Now consider the two sets of vertices: $\{v; l_v < x\}$ and $\{v; l_v \geq x\}$ whose union is V_1. By assumption $x \neq l$, both sets are non-empty. Note that for the first set, any vertex must have $r_v < x$ otherwise we may pick that vertex. But then the support of the first set is contained in $[l, x-1]$ and the second is $[x, r]$, contradicting lemma 4.6.2. $\square$

Corollary 4.6.4. *Let V_1 be a connected component of the overlap graph of an interval graph with support $[l, r]$, with more than 1 vertex. Then we may find a sequence of vertices $v_1, v_2, \ldots$ with the following properties: $l = l_{v_1} < l_{v_2} < l_{v_3} \ldots$, and for each vertex: $l_{v_i} < l_{v_{i+1}} \leq r_{v_i}$*

Proof. We repeatedly apply lemma 4.6.3. Stab $[l, r]$ with the value r to obtain the last vertex in the chain v_p. Stab $[l, r]$ with the value l_{v_p} to obtain v_{p-1} with the property that $l_{v_{i-1}} < l_{v_i} \leq r_{v_{i-1}}$, and repeat. $\square$

Next we show how two overlap connected components interact.

Lemma 4.6.5. *Let V_1, V_2 be two connected components of $O(G)$ where G is an interval graph. Let* `support`$(V_1) = [l_1, r_1]$ *and* `support`$(V_2) = [l_2, r_2]$ *Then one of the following is true*

- *$[l_1, r_1]$ does not intersect $[l_2, r_2]$*
- *$[l_1, r_1] \subseteq [l_2, r_2]$ and there exists a vertex $u \in V_2$ such that $[l_1, r_1] \subseteq [l_u, r_u]$. (or the symmetric case)*

Proof. Clearly the two are conditions are mutually exclusive. Thus we will consider the case where $[l_1, r_1], [l_2, r_2]$ intersect. First suppose that they overlap, and without loss of generality, $l_1 < l_2 \leq r_1 < r_2$.

We first note that, if $l_2 = r_1$, then there exists a vertex $v \in V_1$ such that $l_v < r_v = r_1$ and a vertex $u \in V_2$ with $l_2 = l_u < r_u$ by lemma 4.6.3. These two vertices clearly overlap contradicting the fact that V_1 and V_2 are distinct overlap connected components.

Again by lemma 4.6.3, there exists a vertex $v \in V_1$ with $l_v < l_2 < r_v$. We note that if $r_v = l_2$ we again have the same scenario as above, so the vertex chosen must have $l_2 < r_v$. Furthermore, there is an vertex $u \in V_2$ with $l_u < r_2 \leq r_u$. If $l_u \geq r_v$ then these two overlap.

If not, then we stab V_2 with the value l_u to obtain u_1 with $l_{u_1} < l_u \leq r_{u_1}$. Continue in this way until we find some vertex u_i such that $l_{u_i} < r_v < l_{u_{i-1}} \leq r_u$. Such a vertex u_i exists because l_{u_i} forms a decreasing sequence. This u_i overlaps v a contradiction (this is a restricted from of 4.6.4 where we do not start at the end).

Thus $[l_1, r_1], [l_2, r_2]$ cannot overlap and one must be contained in the other. By symmetry, assume that $[l_1, r_1] \subseteq [l_2, r_2]$.

Let $u \in V_2$ be the vertex such that $l_u \leq l_1$ such that r_u is maximized. We will show that $r_u \geq r_1$.

First we note that $l_1 \leq r_u$ as otherwise, no vertex in V_2 contains any point in (r_u, l_1) in its support, a contradiction.

Now suppose that $r_u < r_1$. by corollary 4.6.4, we find a chain of vertices $v_1, \ldots \in V_1$ such that $l_1 = l_{v_1} < l_{v_2} \ldots$. If $r_u = l_{v_i}$ for some i, then u overlaps with v_i. Otherwise, there either exists some i such that $l_{v_i} < r_u < l_{v_{i+1}}$ or $l_v \leq r_u < r_v = r_1$ where v is the last vertex in the chain. But in this case, we either have the inequalities, $l_u < l_{v_i} < r_u < l_{v_{i+1}} \leq r_{v_i}$ and u overlaps with v_i or $l_u < l_v \leq r_u < r_v = r_1$ and u overlaps with v, a contradiction.

Thus u has the property that $l_u \leq l_1 \leq r_1 \leq r_u$ so that $[l_1, r_1] \subseteq [l_u, r_u]$. □

With all of our structural lemmas complete we now give our class of interval graphs by defining our maximal cliques. Fix σ. Let $C_1 = \{1, \ldots \sigma\}, C_2 = \{2, \ldots, \sigma + 1\}$, $C_\sigma = \{\sigma, \ldots, 2\sigma - 1\}$ and finally $C_{\sigma+1} = \{\sigma + 1, \ldots, 2\sigma\}$. For $C_{\sigma+2}$ to $C_{n-\sigma+1}$ we apply the following operation to get C_{i+1} from C_i: choose an arbitrary element of C_i that is in the smaller half of values and remove it, then add the next vertex $(i + \sigma - 1)$. For example, to go from $C_{\sigma+1}$ to $C_{\sigma+2}$ we delete one of the elements in the smaller half, that is one of $\{\sigma + 1, \ldots, (3/2)\sigma\}$, and add the next vertex, which is $2\sigma + 1$. See Figure 4.4.

We will now show that there is at most 2 ways to order the maximal cliques so that it satisfies the consecutive clique property.

Theorem 4.6.6. *Let G be an interval graph obtained in the algorithm above. Then there are at most two orderings of the maximal cliques which satisfy the consecutive clique property.*

Proof. First we note that there are exactly two vertices with the same left and right endpoints: $\texttt{support}(v_1) = [1, 1]$ and $\texttt{support}(v_n) = [n - \sigma + 1, n - \sigma + 1]$. Furthermore, if we remove them and try to re-add them, to maintain the same graph, v_1 must go in the first clique and v_n must go in the last clique. Thus if we show the rest of the graph is

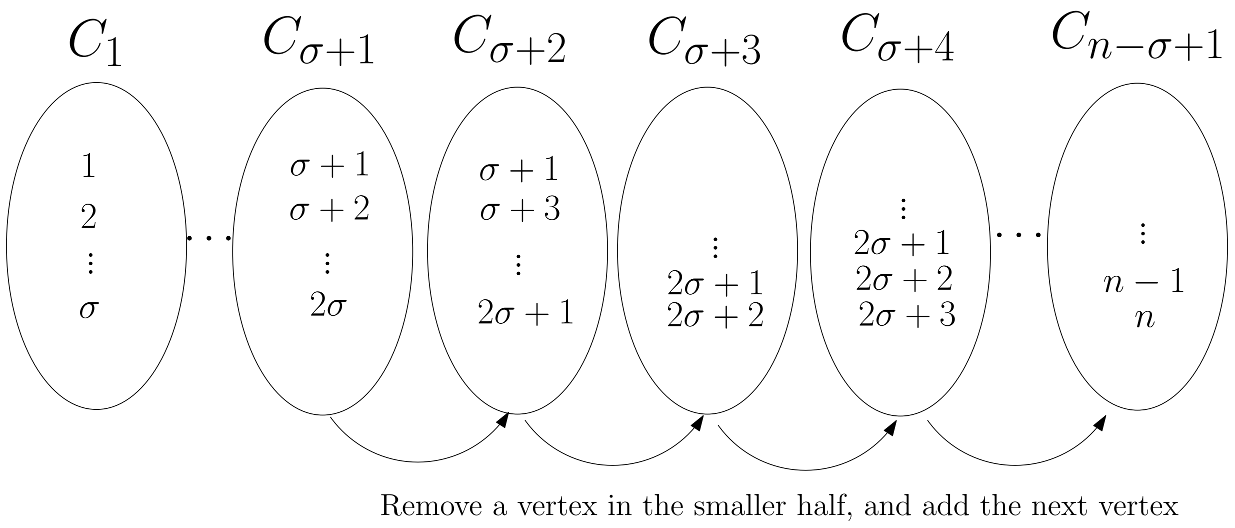

C_1
$C_{\sigma+1}$
$C_{\sigma+2}$
$C_{\sigma+3}$
$C_{\sigma+4}$
$C_{n-\sigma+1}$
1 2 $\vdots$ σ
$\sigma+1$ $\sigma+2$ $\vdots$ 2σ
$\sigma+1$ $\sigma+3$ $\vdots$ $2\sigma+1$
$\vdots$ $2\sigma+1$ $2\sigma+2$
$\vdots$ $2\sigma+1$ $2\sigma+2$ $2\sigma+3$
$\vdots$ $n-1$ n
Remove a vertex in the smaller half, and add the next vertex

Figure 4.4: The construction of our subset of interval graphs. At each maximal clique after $C_{\sigma+1}$, we remove a vertex from the smaller half, and add the next vertex. In this example, to obtain $C_{\sigma+2}$ from $C_{\sigma+1}$ we remove the vertex $\sigma+2$ in the smaller half, and added $2\sigma+1$

overlap connected, then by theorem 4.6.1 there are at most two ways to order the cliques and satisfy the consecutive clique property.

Consider any component V_1 with support $[l, r]$. We note that $l \neq r$ since no vertex has the same left and right endpoints any more. We claim that $l = 1$ and $r = n - \sigma + 1$. If $r \neq n - \sigma + 1$, then consider the vertex $v_{r+\sigma-1}$ which is added in C_r. Since we do not remove it this clique, $r_{v_{r+\sigma-1}} > r$. Furthermore, since $\texttt{support}(V_1) = [l, r]$, there exists a vertex $v \in V_1$ with $l_v < r_v = r$. These two vertices overlap so $v_{r+\sigma+1} \in V_1$ and $\texttt{support}(V_1) \neq [l, r]$. To show that $l = 1$ is similar by considering the vertex that ends in C_l, which must overlap the vertex $v \in V_1$ that has $l = l_v < r_v$.

Now suppose that $G \setminus \{v_1, v_n\}$ were not overlap connected, and let V_1, V_2 be two overlap connected components. By above, we must have $\texttt{support}(V_1) = \texttt{support}(V_2) = [1, n-\sigma+1]$. By lemma 4.6.5, there exists a vertex $u \in V_2$ such that $[1, n-\sigma+1] \subseteq \texttt{support}(u)$. However, by construction, no vertex that starts in C_1 can end in $C_{n-\sigma+1}$ a contradiction.

Therefore $G \setminus \{v_1, v_n\}$ is overlap connected and we are done. □

With this we are able to show lower bounds for many classes of interval graphs. First we give an alternate lower bound proof for interval graphs of Acan et al. by setting $\sigma = n/\lg n$.

Theorem 4.6.7. *To represent interval graphs, we need $n \lg n - O(n \lg\lg n)$ bits.*

Proof. The number of graphs that can be created using the above algorithm is the following. We make choices at $n - 2\sigma$ cliques - to determine which vertex should end at that clique. Each of these choices is between $\sigma/2$ elements. Thus the number of graphs we obtain is $(\sigma/2)^{(n-2\sigma)}$. Hence the information theoretic lower bound is:

$$\lg{(\sigma/2)^{(n-2\sigma)}} = (n-2\sigma)\lg(\sigma/2)$$

Setting $\sigma = n/\lg n$ completes the proof. □

Next we will apply this construction to improve the lower bound results of [18] on bounded degree and bounded chromatic number interval graphs.

Theorem 4.6.8. *To represent interval graphs with bounded chromatic number σ, we need $n\lg\sigma - O(\sigma\lg\sigma + n)$ bits. (or equal to $n\lg n - o(n\lg n)$ when $\sigma \geq n/\lg n$)*

Proof. As interval graphs are perfect graphs, the chromatic number is equal the size of the maximum clique. By construction, all graphs G has maximum clique size σ. Thus the lower bound is simply

$$(n-2\sigma)\lg(\sigma/2) = n\lg\sigma - O(\sigma\lg\sigma + n)$$

Finally we note that this only applies when $\sigma = O(n/\lg n)$ as larger σ (especially when $\sigma = \Theta(n)$) would cause the lower order term to cancel out with the leading term. But as we have shown that the lower bound is already $n\lg n - o(n\lg n)$ for $\sigma = n/\lg n$, any larger σ would also inherit this lower bound. □

Theorem 4.6.9. *To represent interval graphs with bounded degree σ, we need $n\lg\sigma - o(n\lg\sigma)$ bits.*

Proof. First rename the size of the maximal cliques in our construction as Σ to avoid variable name collisions.

To limit the degree of our vertices, we need to make sure that the clique that any vertex ends on is not too far from the clique that the vertex begins. For a vertex v with $\texttt{support}(v) = [l_v, r_v]$, we have $\deg(v) = (\Sigma - 1) + (r_v - l_v)$. The first term says that it is adjacent to $\Sigma - 1$ vertices of the clique that it begins at, and the second term says that for each additional clique, it gains one more neighbour.

To limit the size of $r_v - l_v$, rather than selecting an arbitrary vertex to remove, every Δ cliques, we always remove the smallest vertex. Consider the rank of any vertex v in the cliques that contain it (that is, how many vertices are smaller than it in the clique?). In the clique that it begins at, it is rank Σ - the largest numbered vertex. Since we always remove a vertex in the smaller half, the next $\Sigma/2$ cliques will decrease its rank by 1. Since we remove the smallest vertex in the clique every Δ cliques, we are guaranteed to decrease the rank of v every Δ cliques and thus remove it after at most $\Delta \cdot \Sigma/2$ cliques. Thus $r_v - l_v \leq \Sigma/2 + \Delta\Sigma/2$ and $\deg(v) \leq \Sigma/2 + \Sigma/2 + \Delta\Sigma/2 = \frac{\Sigma}{2}(2+\Delta)$. And thus we set $\sigma = \frac{\Sigma}{2}(2+\Delta)$.

The number of graphs we obtain is also changed. Rather than making $(n - 2\Sigma)$ choices, we make $\frac{\Delta-1}{\Delta}(n-2\Sigma)$ choices since we always remove the smallest vertex every Δ cliques. Thus the lower bound is instead:

$$\frac{\Delta - 1}{\Delta}(n - 2\Sigma)\lg(\Sigma/2)$$

By setting $\Sigma = 2\sigma/\lg\sigma$ and $\Delta = \lg(\sigma) - 2$, the lower bound we obtain is

$$n\lg\sigma - O(n)$$

□

4.7 Discussion

In this chapter, we use the results of Chapter 3 to construct succinct distance oracles for static interval graphs in $n\lg n + O(n)$ bit of space and $O(1)$ query times, and proper interval graphs using $2n + o(n)$ bits of space and $O(1)$ query times. We also abstracted the solution so that it can be used easily for other subclasses of graphs to only support the retrieval of left/right endpoints of intervals.

We also used the reduction of the distance problem from circular arc graphs to distance graphs to give succinct distance oracles for them as well, using $n\lg n + o(n\lg n)$ bits of space and $O(1)$ query times.

We used this abstract solution to give data structures for bounded chromatic number interval graphs, a succinct data structure occupying $n\lg\sigma + o(n\lg\sigma)$ bits of space and $\sigma\lg n$ query times and a compact data structure using $2n\lg\sigma + o(n\lg\sigma)$ bits of space and $O(\lg\lg\sigma)$ query times.

Lastly we showed that the data structures for both bounded degree and bounded chromatic number interval graphs are indeed succinct by proving $n\lg\sigma$ lower bounds, improving on the previous results by a factor of 6.

Some problems left open are mainly for bounded chromatic number interval graphs. Our succinct data structure has relatively slow query times of $O(\sigma \lg n)$, and thus it would be of interest to reduce it, if not to $O(1)$, then reduce the dependence on σ to $\lg \sigma$. Our compact data structure's query time can also potentially be reduced to $O(1)$ rather than $O(\lg \lg \sigma)$.

In the case of circular arc graphs, the two succinct data structures both have flaws. In one, `degree` cannot be implemented at all other than naively searching for all the neighbours and then counting, using $O(\texttt{degree} + 1)$ time. On the other hand, the data structure computing `degree` in a more clever way imposes a $O(\lg n / \lg \lg n)$ overhead on all other queries.

Chapter 5

Beer Path Queries in Interval Graphs

In this chapter, we will consider a variant of the distance query. In `distance` we are looking for the length of a shortest path between two vertices. In `beer_distance`, in addition to our graph (in general potentially weighted, though we will consider unweighted graphs), some of the vertices are designated as beer vertices. The shortest paths (between two vertices) considered in the query must pass through at least one of the given beer vertices.

This chapter is organized as follows: we being with a brief review of relevant previous works in Section 5.1, and an review of existing results in Section 5.2. We will first study the problem in proper interval graphs in Section 5.3, then generalize it to interval graphs in Section 5.4. We will then consider the problem of enumerating beer interval graphs in an attempt to prove a lower bound in Section 5.5. Finally, we will summarize the results and give future areas of consideration in Section 5.6.

5.1 Previous Work

The concept of a beer path was recently introduced by Bacic et al. [7] in 2021[1]. The premise is simple, suppose you wish to visit a friend, and wish to pick up some beer along the way because you don't want to show up empty handed, what is the fastest way to do so? More formally, for a graph, we specify a set of vertices, which will act as beer stores. A beer path is one which passes through at least one of these designated vertices. We will say a beer graph is one where we have designated a subset of the vertices to be beer stores.

[1]Though the problem is very natural, the routing problem with these constraints does not appear to have been studied until very recently.

Though this premise may be somewhat silly, it can have many applications. For example, suppose you are going on a road trip and to be efficient, want to drop something off at a post office on the way. Or perhaps on your trip, you realize that you currently don't have enough gas, so you must visit a gas station somewhere along the way. Another hypothetical situation would be if a package needs to be transported, but due to regulations, one of the stops must be equipped for an inspection. In Bacic et al. [7], they showed that on an outerplanar graph of n vertices, a data structure of size $O(m)$ words for any $m \geq n$ can be constructed in $O(m)$ time to support shortest beer path and beer distance - the length of the shortest beer path in $O(\alpha(m, n))$ time, where α is the inverse Ackermann function.

Although a shortest beer path may not be simple, it will always consist of two shortest paths: from the beer store to the source, and to the destination.

5.2 Preliminaries

In a beer graph, we take any underlying graph G together with a set $B \subseteq V$ of *beer vertices*. If G belongs to a class of graphs of interest, we will use the prefix "beer" to denote that we have also been given the set B of beer vertices. For example, if G is an interval graph, then corresponding graph with beer vertices is a *beer interval graph*.

Definition 5.2.1. A ***beer graph*** is a tuple (G, B) consisting of a graph and a set of beer vertices $B \subseteq V$.

Definition 5.2.2. If (G, B) is a beer graph, then we are interested in these queries:

- `beer_spath` (u, v): return a shortest path between the vertices u and v such that at least one of the beer vertices appears on the path.
- `beer_distance` (u, v): return the length of the shortest path between vertices u and v such that at least one of the beer vertices appears on the path.

These are the restricted queries to the ordinary `spath` and `distance` queries, which do not have the constraint that it must pass through a beer vertex.

For example, if $B = V$, then the two queries reduces to ordinary shortest path or distance in the graph (this is also the case if either u or v is a beer vertex). On the other extreme, if $B = \{b\}$ is a singleton, then the query reduces to two ordinary shortest path or distance queries in the graph.

5.2.1 Interval Graphs

We will briefly review the definition of an interval graph here, and restate the two theorems proved in Chapter 4.

An interval graph G is a graph where we may assign an interval on the real line to each vertex v: $I_v = [l_v, r_v]$ such that two vertices u, v are adjacent exactly when the corresponding intervals intersect [40]. In particular, we may sort the endpoints so that the values are integers between 1 and $2n$. We sort the vertices based on their left endpoints, so that when we refer to vertex v, we are referring to a number (the rank of the vertex in the sorted order), and thus, a statement such as $u < v$ makes sense.

Theorem 4.4.3 (Succinct interval graphs with distance)**.** *An interval graph G can be represented using $n \lg n + (5+\epsilon)n + o(n)$ bits to support* `adjacent`*,* `degree` *and* `distance` *in $O(1)$ time,* `neighborhood` *in $O(\texttt{degree}(v)+1)$ time, and* `spath`(u, v) *in $O(\texttt{distance}(u, v)+1)$ time.*

An interval graph G is *proper* (or a *proper interval graph*) if we can choose the intervals corresponding to vertices such that no two intervals are nested.

Theorem 4.4.8 (Succinct proper interval graphs with distance)**.** *An 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 $O(\texttt{distance}(u, v))$ time.*

5.2.2 Dyck Paths

For our lower bound, we will be discussing Dyck paths as one of the most well known class of combinatorial objects satisfying the catalan numbers recurrence [79]. A Dyck path (on the 2D plane) of length $2n$ is a path from $(0, 0)$ to $(2n, 0)$ using $2n$ steps, n of which are $(1, 1)$ steps which are referred to as up-steps, and n of which are $(1, -1)$ steps and are referred to as down-steps. Such a path must also satisfy the condition that it never reaches below the x-axis. It is well known that the number of Dyck paths of length $2n$ is $C_n = \frac{1}{n+1}\binom{2n}{n}$ the n-th Catalan number.

A Dyck path that never touches the x-axis except at the start and end, is referred to as an irreducible Dyck path. By removing the up-step at the beginning and the down-step at the end, the remainder of the path is simply a Dyck path of length $2(n-1)$. Thus the number of irreducible Dyck paths of length $2n$ is simply C_{n-1}.

For any Dyck path, we may associate an up-step with an open parenthesis (and a down-step with a close parenthesis). The sequence we obtain from a Dyck path is a balanced parenthesis sequence (and vice versa) as the Dyck path condition is exactly the condition that the excess in the balanced parenthesis sequence is never negative (i.e each closing parenthesis has a matching opening parenthsis). This well known bijection allows us to associate an forest to each Dyck path, using the well known bijection for forests and balanced parentheses (via a depth-first traversal) [61]. In particular, if the Dyck path were irreducible, then the forest is just a single tree.

5.2.3 Succinct Data Structures

Recall that information theoretic lower bound to represent a family of objects with N elements is $\lceil \lg N \rceil$ bits. Any fewer bits and we do not have enough bit strings to assign a unique one to each object, and thus cannot distinguish between them. A succinct data structure aims to use $\lg N + o(\lg N)$ bits to represent these objects while supporting the relevant queries.

We will once again require the various forms bitvectors. Recall:

Lemma 2.3.2 ([62]). *A bit-vector of length n can be represented in $n + o(n)$ bits to support* `access`, `rank`, `select` *in $O(1)$ time.*

Lemma 2.3.4 ([69]). *For a bit-vector of length n, containing m 1-bits*[2], *and any constant c, there is a data structure using*

$$\lg \binom{n}{m} + O\left(\frac{n}{\lg^c n}\right) \le m \lg \left(\frac{n}{m}\right) + O\left(\frac{n}{\lg^c n} + m\right)$$

bits of space that supports `access`, `rank`, `select` *in $O(1)$ time.*

We will also require the result from Chapter 3 on ordinal trees. The operations we will use will mainly be conversions between the nodes' numbers in the different traversals: pre-order, post-order, level-order. These operations can be done in $O(1)$.

Theorem 3.5.2 (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 2.1 in $O(1)$ time.*

We will be using permutations as well.

[2]The number of such bit-vectors is $\binom{n}{m}$.

Lemma 2.3.8 ([60])**.** *Let P be a permutation. Then we may represent P using $(1 + 1/f(n))n \lg n + o(n \lg n)$ bits to support the computation of P and P^{-1} in $O(f(n))$ time ($1 \leq f(n) \leq n$). In particular, if we set $1/f(n) = \varepsilon$ for constant $\varepsilon > 0$, then the space is $(1 + \varepsilon)n \lg n + o(n \lg n)$ bits and the time is $O(\frac{1}{\varepsilon}) = O(1)$.*

We will need the various forms of orthogonal range queries.

Lemma 2.5.3 ([16])**.** *Let S be a set of points from the universe $M = [1..n] \times [1..n]$, where $n = |S|$. S can be represented using $n \lg n + o(n \lg n)$ bits to support orthogonal range counting in $O(\lg n / \lg \lg n)$ time, and orthogonal range reporting in $O(k \lg n / \lg \lg n)$ time, where k is the size of the output.*

Lemma 2.5.7 ([68])**.** *For n 3D points and constant $\varepsilon > 0$, we may support 5-sided orthogonal reporting queries using $O(n \lg n)$ bits of space and $O(k \lg^{\varepsilon} n)$ time or $O(n \lg n \lg \lg n)$ bits of space and $O(k \lg \lg n)$ time, where k is the size of the output.*

We may also achieve the same complexities for emptiness as well.

And finally, predecessor queries:

Lemma 2.5.5 ([83])**.** *There is a data structure for the predecessor/successor problem that uses $O(n)$ words of space and query time $O(\lg \lg |U|)$.*

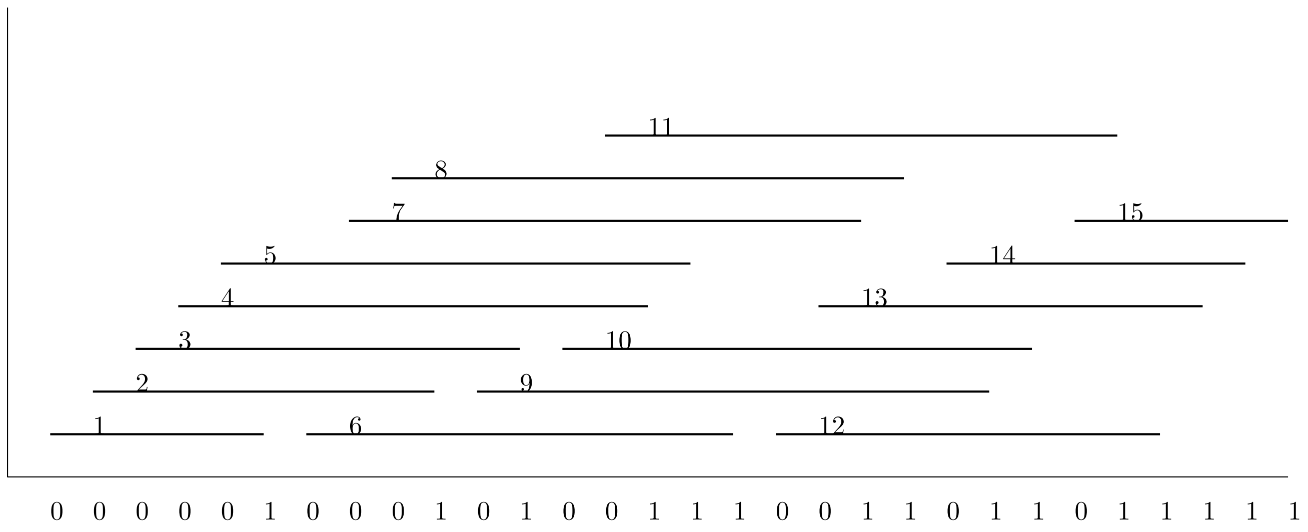
5.3 Proper Interval Graphs

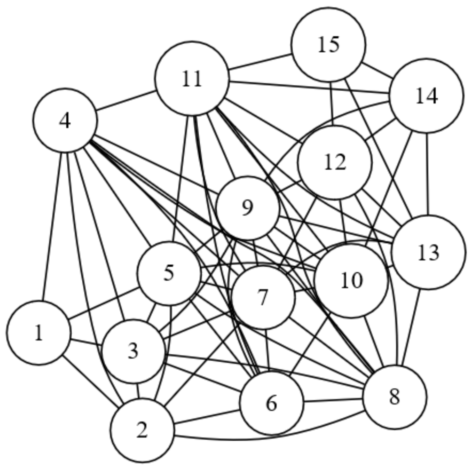
In this section we investigate beer paths in proper interval graphs. We will base our data structure on Theorem 4.4.8, and modify it to account for beer vertices.

We begin with an example:

Example 5.3.1. Consider the graph with the interval representation given by the bit string: 000001000101001110011011011111 (recall the bit-vector B in Section 4.3.1 as we were describing the data structure of Acan et al. Each 0 encodes the left endpoint of some interval, and each 1 encodes the right endpoint of some interval. Since the graph is proper, the endpoints of the ith vertex must be the indices i-th 0 and the i-th 1.). This gives the vertex 1 a left endpoint at coordinate 1 and right index at coordinate 6 (the first 0 and first 1 in the sequence respectively). See Figure 5.1.

Consider the shortest path between nodes 13 and 3. By Lemma 4.4.2, a shortest path is $13 \rightarrow 7 \rightarrow 3$. The problem would be easy if one of these nodes were a beer node, say





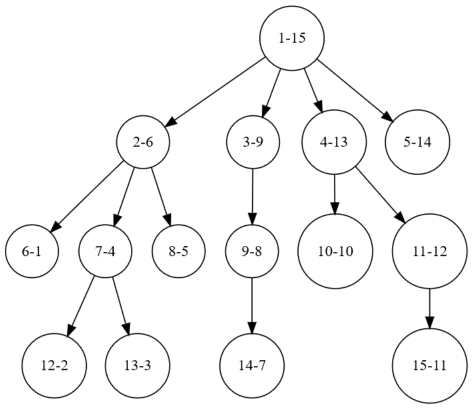


Figure 5.1: An proper interval graph corresponding to the bitvector 000001000101001110011011011111, and the corresponding distance tree. Each node of the tree is labelled with its rank in level-order and post order traversals.

node 7 then there would be far less to do. However consider the case that the only beer node were node 6, then a `beer_spath` would be $13 \to 7 \to 6 \to 3$. On the other hand, if there were also a beer node at node 8, then we can take the path $13 \to 8 \to 3$.

5.3.1 Calculating Beer Distance

Let us recall that in Theorem 4.4.8, we represented the proper interval graph G using a distance tree T. There is a bijection between the vertices of the graph G and the nodes of the tree - vertex v is mapped to the vth node in the tree in level-order by Lemma 4.4.1. Thus by v we will simultaneously refer to the vertex and the node in the distance tree. As the conversion between level-order ranks, pre-order ranks and post-order ranks in a tree are all $O(1)$ (Theorem 3.5.2) (and typically, nodes in a tree are referred by their pre-order ranks), we will implicitly convert between them as the situation requires. All the queries are reduced to tree operations and can be done in $O(1)$ time.

Let us recall Lemma 4.4.2. The distance between two vertices u, v with $\texttt{depth}(u) \leq \texttt{depth}(v)$ is found by taking the node to root path from v. As we will need to refer to these nodes, instead of naming them $v = v_1, \dots v_k = r$, instead let $k_2 = \texttt{depth}(v)$, and name the nodes $v = v_{k_2}, \dots v_0 = r$, so that the subscript denotes the depth of the node on the path. Let $\texttt{depth}(u) = k_1$.

We note that this is only one of many possible shortest paths. To accommodate the beer vertices, we will investigate what all the possible shortest paths might look like. To this end, we will consider the following question: let $u < w < v$, is w on a shortest path between u and v? Equivalently, is $\texttt{distance}(u, v) = \texttt{distance}(u, w) + \texttt{distance}(w, v)$?

Definition 5.3.2. Given two vertices u, v, a vertex w with $u < w < v$ ***preserves the distance*** (w.r.t. u, v) if $\texttt{distance}(u, v) = \texttt{distance}(u, w) + \texttt{distance}(w, v)$.

Let $\texttt{node\_rank}_{\texttt{POST}}(v)$ denote the post-order rank of a node in the tree. It is clear that if $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$, then u is to the left of v_{k_1}, so that $u < v_{k_1}$, and similarly for the reversed inequality. Finally, we note that for nodes on the same level of the tree, their post-order numbers are sorted. That is if $u < v$ are on the same level, then $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$ (and vice versa).

Lemma 5.3.3. *Let $u < w < v$ be 3 nodes in a proper interval graph. In the case that $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$, then w is on a shortest path exactly when either $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$ or $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$ (that is w*

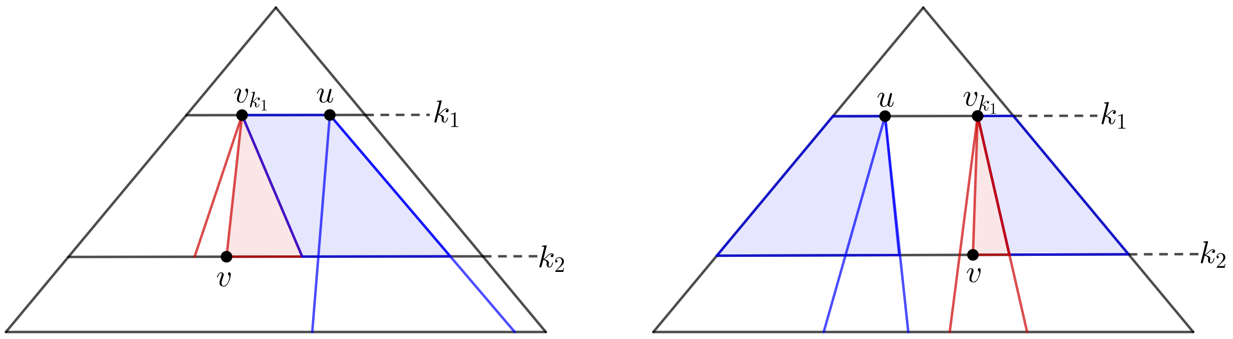


Figure 5.2: The union of the two shaded regions capture the nodes which preserve the distance in Lemma 5.3.3. The left is the case when $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v)$, and the right is the case when $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. The blue region represents complete subtrees that are included, while the red represents the nodes to the right of the path to the root from v, used in subsection 5.3.3. Note the nodes on level k_1 to the left of u and on level k_2 to the right of v are not included.

preserves the distance). In the case that $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v)$, *then w preserves the distance exactly when* $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$. *Furthermore, if w does not preserve the distance, then the path passing through w increases the distance by 1. That is* $\texttt{distance}(u, v) + 1 = \texttt{distance}(u, w) + \texttt{distance}(w, v)$.

Proof. First we consider the case when $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. By our previous remark, this implies that $v_{k_1} > u$ (that is v_{k_1} is to the right of u on level k_1). Furthermore, this implies that $\texttt{distance}(u, v) = k_2 - k_1 + 1$.

For each level $k_1 < k_3 \leq k_2$, we consider the nodes in level-order between v_{k_3} and v_{k_3+1} and denote them as $V_{k_3} = \{w; v_{k_3} \leq w < v_{k_3+1}\}$. These are the nodes that are adjacent to v_{k_3+1} (that are before it in level order) and thus have a distance $k_2 - k_3$ from v. You can see this as a shortest path produced by our `spath` query would be $w, v_{k_3+1}, \ldots, v$. This set contains nodes from two levels in the tree: k_3 and $k_3 + 1$. First consider the nodes w on level k_3. These nodes are exactly those that $\texttt{node\_rank}_{\texttt{POST}}(w) \geq \texttt{node\_rank}_{\texttt{POST}}(v_{k_3}) > \texttt{node\_rank}_{\texttt{POST}}(v)$. If we look at a shortest path from w to u, we see that the chain we produce $w_{k_1}, \ldots, w_{k_3}$, has the property that $w_{k_1} \geq v_{k_1} > u$. This is because as $\texttt{node\_rank}_{\texttt{POST}}(w_{k_3}) > \texttt{node\_rank}_{\texttt{POST}}(v_{k_3})$, and this inequality is preserved as we repeatedly take the parent operation on both chains. Therefore, the distance between w and u is $k_3 - k_1 + 1$, and w preserves the distance.

Next consider the nodes on level $k_3 + 1$. These are the nodes with $w < v_{k_3+1}$, and equivalently, exactly those on this level that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(v)$. As above, we consider the path from w to u, which expands out as the chain $w_{k_1}, \ldots, w_{k_3+1}$. In the case that $w_{k_1} \leq u$, the distance is $k_3 - k_1 + 1$, and if $w_{k_1} > u$, then the distance is $k_3 - k_1 + 2$. Thus we see that w preserves the distance when $w_{k_1} \leq u$. This implies that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$, as either $w_{k_1} = u$ and u is the last node in its subtree by post-order numbers, or w_{k_1} is to the left of u and that relation is preserved down the chain. Thus in the set V_{k_3}, the nodes that preserve the distance are exactly those with $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$ or $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$. Taking the union of the V_k and we see that this is exactly the condition for a node to preserve the distance. Furthermore, the nodes that do not preserve the distance only increase the distance by 1 ($\texttt{distance}(u, v) + 1 = \texttt{distance}(u, w) + \texttt{distance}(v, w)$).

The second case is when $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(u)$. In this case, we have $v_{k_1} \leq u$ and thus $\texttt{distance}(u, v) = k_2 - k_1$. Again we consider sets V_{k_3} and the nodes on the levels k_3 and k_3+1. These nodes have a distance of $k_2 - k_3$ to v. First we consider the nodes on k_3+1. As in the previous case, the distance must be either $k_3 - k_1 + 1$ or $k_3 - k_1 + 2$, but in either case, we cannot preserve the distance. In fact, since $\texttt{node\_rank}_{\texttt{POST}}(v_{k_3+1}) \leq \texttt{node\_rank}_{\texttt{POST}}(u)$, we see that $w_{k_1} \leq v_{k_1} \leq u$, hence the distance must actually be $k_3 - k_1 + 1$.

Next we consider the nodes on level k_3. Expanding out the path, we have two cases: either $w_{k_1} \leq u$ or $w_{k_1} > u$. In the first case, the distance is $k_3 - k_1$ and these nodes preserve the distance, and the second case the distance is $k_3 - k_1 + 1$. The condition for $w_{k_1} \leq u$ is $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$, as in this case, either w is in the subtree rooted at u or in the subtree rooted at a node to the left of u on level k_1.

Combining the cases we see that the w preserves the distance exactly when $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$. And if w does not, it only increase the distance by 1. $\square$

Remark: when u and v are on the same level, then by Lemma 4.4.6, they are adjacent. Thus no nodes can preserve the distance.

See figure Figure 5.2 for a pictorial representation of the criteria. Now we can describe the process of determining the beer distance. The idea is to cover the nodes of the tree with 3 sets, and determine the best possible beer distance using beer vertices in each of the 3 sets (if a set has no beer vertices, then the distance will naturally be ∞). Finally we take the minimum of the 3. We will call the best vertex in each set a *candidate*.

First we note that if either $u, v \in B$, then we do not need to do anything and simply return $\texttt{distance}(u, v)$ (or $\texttt{spath}(u, v)$).

Candidate 1: The set of vertices is $\{w \in B; w > v\}$. We claim that the best beer vertex (i.e. the shortest beer path through this vertex is the shortest beer path possible among all that go through a beer vertex from this set) in this set is the smallest one. To show this, we will use the following lemma.

Lemma 5.3.4. *Let $u < v < w$ be 3 vertices in a proper interval graph, then* $\texttt{distance}(u, v) \leq \texttt{distance}(u, w)$. *By symmetry,* $\texttt{distance}(v, w) \leq \texttt{distance}(u, w)$.

Proof. Let the depths be $\texttt{depth}(u) = k_1, \texttt{depth}(v) = k_2, \texttt{depth}(w) = k_3$ with $k_1 \leq k_2 \leq k_3$. We consider the chain from w: $w_{k_1}, \dots, w_{k_3} = w$. Since $v < w$, there exists an index i such that $w_{i-1} < v \leq w_i$. and $i \leq w_3$. Hence we may create the path starting from v as $v \to w_{i-1} \to w_{i-2} \cdots w_{k_1}$, which is a path to u of at most the length as the path from w. The result follows. $\square$

The vertex in $\{w \in B; w > v\}$ that minimizes the value of $\texttt{distance}(u, w) + \texttt{distance}(v, w)$ is of course the w of minimal index.

Candidate 2: The set of vertices $\{w \in B; w < u\}$. We take the largest vertex in the set as the candidate using Lemma 5.3.4.

Candidate 3: The set of nodes $\{w \in B; u < w < v\}$. By Lemma 5.3.3, we need to determine whether there exists a node such that either

$$\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(v)$$

or

$$\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u) \text{ or } \texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$$

depending on $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. If such a vertex exists, then it is the candidate, with distance $\texttt{distance}(u, v)$. If no such node exists but the set is non-empty, we may take any vertex in the set as the candidate, with distance $\texttt{distance}(u, v) + 1$.

5.3.2 First Data Structure For Beer Distance

Here we discuss how to use the previous results to create a data structure for the queries. In this subsection, we discuss a relatively simple data structure which has decent run times. However, the space is dependent on $|B|$, the number of beer nodes, and in the case that $|B| = \Theta(n)$ is large, the space bound is also unacceptably large. In the next subsection, we will show how to remove this dependence on $|B|$ at the cost of slightly worse run times.

To store the beer vertices, we store a bit vector B of length n so that $B[i] = 1$ if the ith vertex is a beer vertex. This uses $n + o(n)$ bits of space. As in the previous subsection we will assume that neither u nor v are beer vertices (and we can check by looking at $B[u], B[v]$).

Candidate 1: We use $\texttt{rank}(v)$ to find how many beer nodes are up to v. The smallest beer node that is larger than v can be found using $\texttt{select}(\texttt{rank}(v) + 1)$.

Candidate 2: Similarly to candidate 1, we find it by $\texttt{select}(\texttt{rank}(u))$.

Candidate 3: For every beer node w, we store it in a 2D range emptiness data structure using the coordinates $(w, \texttt{node\_rank}_{\texttt{POST}}(w))$ (by our notation w is just the level-order number of the node w). In the case that $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$, we need the nodes such that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$ or $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$ and $u < w < v$. This is translated to the rectangles
$(u, v) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(u))$ and $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \infty)$.

For the second case when $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v)$, we need the nodes that $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$. This is the rectangle $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \texttt{node\_rank}_{\texttt{POST}}(u))$.

This suffices to determine the distance. To list out the path, we first determine which candidate to use. Candidates 1 and 2 can simply list out the path using two `spath` queries. For candidate 3, it either preserves the distance or does not. If it does not, then any beer node in the range will suffice, and we list out the path using two `spath` queries. Otherwise if there is a node that preserves the distance we list out the path between u, v one step at a time, and, at each step, we consider the set V_k, which is an interval in level order. We note that the nodes preserving the distance is a prefix of this interval. Thus we find the first beer vertex in V_k, and check if it preserves the distance, if so add it to the path and list out the path from there. Otherwise, we continue to the next level. Since the best candidate is candidate 3 and it preserves the distance, we are guaranteed that somewhere along this path, we will find a beer node preserving the distance.

Furthermore, as listing out the path for Candidate 3 is at most $O(\texttt{distance}(u, v))$ time, we may do this whenever $\texttt{distance}(u, v) = O(\lg^\epsilon n)$ rather than spending the time on the orthogonal range search in the distance query. Thus we have the following theorem:

Theorem 5.3.5. *A beer proper interval graph G can be represented using $3n + o(n) + O(|B| \lg n)$ bits to support the interval graph queries plus* `beer_spath` *in $O(1)$ time per vertex on the path and* $\texttt{beer\_distance}(u, v)$ *in $O(\min(\lg^\epsilon n, \texttt{distance}(u, v)))$ time. If we increase the extra space to $O(|B| \lg n \lg \lg n)$ bits, we may support* $\texttt{beer\_distance}(u, v)$ *in $O(\min(\lg \lg n, \texttt{distance}(u, v)))$ time instead.*

Proof. The space required is the distance tree from Theorem 4.4.8, a single length n bit vector for the beer nodes, and a single 2D range emptiness data structure. $\square$

We note that if there are many beer vertices, so that $|B| \in \Theta(n)$, then the space usage would be $\Theta(n \lg n)$. However if $|B|$ were small (ex. $|B| = O(n/\lg^2 n)$), then this data structure will only use $3n + o(n)$ bits.

5.3.3 Improved Data Structure for Beer Distance

Here we show how to improve the space usage of our specialized ranged emptiness query, so that it no longer has any dependence on $|B|$. Since we have the tree structure, the range emptiness can be reduced to checking whether a certain set of tree nodes have any beer nodes in them. In particular, as seen from the previous subsection, we need to support the following rectangles using $o(n)$ bits:

1. $(u, v) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(u))$
2. $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \infty)$
3. $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \texttt{node\_rank}_{\texttt{POST}}(u))$.

We will call these type 1,2 and 3 rectangles.

To make our notation cleaner, we will use the depth of a node in the first coordinate of a rectangle. This means to include all the nodes on that level. To accomplish this, we simply find the first node on that level (in $O(1)$ time) and substitute its level-order number as the value to be used in the rectangle; similarly use the last node of a level for the right end point of the rectangle.

Fix $\Delta = \omega(1)$, and choose an index $1 \leq i \leq \Delta$ such that the number of nodes on levels $k = i \mod \Delta$ is minimized. [3]

We will call these levels *selected levels*, and the nodes on them *selected nodes*. Thus the number of selected nodes is $O(n/\Delta)$ by the pigeonhole principle. For each of these nodes, consider the subtree rooted at them that extends down to the next selected level. We will build the contracted tree T' with these subtree as nodes, and the appropriate edges. A node in T' is a beer node if at least one of the original nodes in the corresponding subtree *except the root* is a beer node. The intuition behind leaving out the root in our definition

[3]This is the same slabbing idea from Chapter 3, not surprising as we are dealing with level order trees.

here is because in our rectangles, both the first and last levels are not full. That is we only consider the nodes with post order number greater than $\texttt{node\_rank}_{\texttt{POST}}(u)$ on the first level, and those with post order numbers less than $\texttt{node\_rank}_{\texttt{POST}}(v)$ on the last level. If we do not leave out the root, then we cannot support these partially filled levels in our range search.

We use a bit vector C to store which nodes in level order are selected. As the number of nodes is $O(n/\Delta) = o(n)$, this compressed bit-vector uses $o(n)$ bits of space. We store the contracted tree T' succinctly, using $2n/\Delta + o(n) = o(n)$ bits. The bit vector B' to mark which nodes of T' are beer nodes is also $n/\Delta = o(n)$ bits. Thus the total space for our contracted tree is $o(n)$ bits.

To support these rectangles, we first reduce the general case to one where both u, v are on a selected level.

Lemma 5.3.6. *We may assume that the inputs u, v are on selected levels at the cost of $O(\Delta)$ extra time.*

Proof. For v, we move up the tree using parent as in the `beer_spath` query. On level k, we take the all the nodes on that level, and find 1) the first beer node, 2) the first beer node after v_k. If the first beer node w has the property that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$, we may answer the type 1 rectangle query as true immediately. If there exists a beer node after v_k, we may answer the type 2 query as true immediately. If the beer node w after v_k has the property that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$, then we may answer the type 3 query as true immediately. Thus we assume we find no beer nodes that will allow us to answer the query immediately. Since each step takes $O(1)$ time, it takes $O(\Delta)$ time to reach a node that is on a selected level.

For u, we move down the tree T'. The essence of the rectangles is that for each level, we wish to split the nodes on that level into 2: those with post-order numbers less than or equal to u, and those greater. Thus as we move down the tree T' we wish to find the node that splits the levels in the same way as u.

By the properties of post-order traversal, the node on the next level that has this property is the largest post-order numbered node less than u. If u has any children, this is the last child of u. If u has no children, this is the last child of the previous internal node (in level-order) from u. As shown by He et al. [43] this is the largest neighbour of u. We will call these nodes u_k for the node on level k. In the case that there are no nodes that satisfy the criteria (that is every node on the next level have a post-order number larger than u), then as no node to the left of u has any children, we must necessarily have that type 1 and type 3 rectangles are empty. As type 2 rectangles do not use $\texttt{node\_rank}_{\texttt{POST}}(u)$ and only its depth, we may choose any node on the closest selected level.

As we descend down the tree, we again take all the nodes on the level, and find 1) the first beer node before u - or equivalently, the first beer node before the node u_k, 2) the last beer node. For the first kind w, if one exists, we may answer type 1 rectangles as true. We also check that it satisfies $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$, and if so answer type 3 rectangles as true. The second kind, we check that $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$ and if so, answer type 2 rectangles as true. □

We will now assume that both u, v are on selected levels. To deal with these queries, we will build the 2D range emptiness query on our contracted tree T' and the nodes on the selected levels in the same way. That is we build two 2D range emptiness query data structures. One on the contracted tree T', and a second on the original tree T, but the points will only come from beer nodes that are also selected nodes. We wish to convert as much of the query to the 2D range emptiness on the contracted tree as possible. We will denote the corresponding node in the contracted tree to u by u'. We note that $\texttt{rank}_C(u) = \texttt{node\_rank}^{T'}_{\texttt{LEVEL}}(u')$ so that we may convert between u and u' easily.

Type 1 rectangles: we convert $(u, v) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(u))$ to $[\texttt{depth}(u'), \texttt{depth}(v')) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(u'))$ in the contracted tree and the same rectangle $(u, v) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(u))$ in the selected nodes. We note that in T', we exclude all nodes on $\texttt{depth}(v')$ since in T this includes only the nodes on that level, but in T' this would include the subtrees as well, which extend down. We also change the left endpoint so that we include the subtrees to the left of u, but as we exclude the roots from those subtrees (in our decision to mark them as beer nodes or not), we do not include more nodes in our search than required.

Type 2 rectangles: In the type 2 rectangles, we note that unfortunately, the rectangle does not contain all the nodes in the subtree $v_{\texttt{depth}(u)}$, only those to the right of the path to v. It does however include the entire subtrees of all the nodes to the right of $v_{\texttt{depth}(u)}$. To handle these complete subtrees (and exclude the subtree rooted at $v_{\texttt{depth}(u)}$) we use the rectangle $[\texttt{depth}(u'), \texttt{depth}(v')) \times (\texttt{node\_rank}_{\texttt{POST}}(v_{\texttt{depth}(u')}), \infty)$. Of course we may find $v_{\texttt{depth}(u)}$ using level-ancestor. We again handle the selected nodes using the same rectangle on them $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \infty)$.

Finally, we need to handle the the nodes in the subtree rooted at $v_{\texttt{depth}(u)}$, to the right of the path to v and above the level of v. We start with the entire subtree of $v_{\texttt{depth}(u)}$, whose nodes are an interval in post-order. Then nodes w with $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$ are exactly the ones we want, except that all the subtrees to the right of

v extend down to the bottom of the tree, rather than being cut off at the depth of v. Thus we need to handle them as well. The way to do this in encoded in the lemma below, where $k_1 = \texttt{depth}(u)$.

Lemma 5.3.7. *Let v be a node in a tree T at depth k_2 and v_{k_1} be the ancestor of v at depth k_1, where both k_1, k_2 are selected levels with a fixed $\Delta > 0$. Let T' be the contracted tree as defined above. Then we are able to answer the query: does the rectangle $(v_{k_1}, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \infty)$ in either:*

$O(n/\Delta \lg n) + o(n)$ additional space and $O(\lg n)$ time.

$O(n/\Delta \lg n) + n + o(n)$ additional space and $O(\lg \lg n)$ time.

Here we do not count the space taken by the tree T.

Proof. We count number of beer nodes in this rectangle and if the count is 0, return false, otherwise return true.

We store a bit vector P, where $P[i] = 1$ if the ith node in post-order is a beer node. The number of beer nodes in the subtree of v_{k_1} to the right of v using two rank operations at $\texttt{node\_rank}_{\texttt{POST}}(v_{k_1})$ and $\texttt{node\_rank}_{\texttt{POST}}(v)$. Finally we need to subtract off the number of beer nodes below the subtrees rooted at the selected nodes to the right of v. To do this, at each selected node x, we store the total number of beer nodes in the subtrees to all the selected nodes on the same level $\texttt{depth}(x)$ to the left of x (including x). The number we need to compute is the difference in the number of beer nodes at subtrees to left of v and the last node on $\texttt{depth}(v)$ that is a descendant of v_{k_1}. The space required to store the number of beer nodes in these subtrees is $O((n/\Delta) \cdot \lg n)$.

We note that normally, to explicitly store P, we need $n + o(n)$ bits. As the beer node are stored in B and we can convert between the indices of B and P in constant time, we may forgo storing the bit vector itself, and only store the auxiliary information. Whenever we need a bit of P, we convert the post order number to level-order and use our level order bitvector B instead. As the $\texttt{rank}$ operation is $O(1)$ we thus need at most $O(\lg n)$ bits from the vector P (this occurs when we need $O(\lg n)$ contiguous bits as the key into a lookup table). Thus we may implicitly store P using only $o(n)$ bits, at the cost of $O(\lg n)$ $\texttt{rank}$ query time.

To compute the last node at depth v and is a descendant of $v_{\texttt{depth}(u)}$, we will store the post order numbers of nodes in a predecessor data structure. For each selected level, we store a predecessor structure containing the post order numbers of the selected nodes at that level. The node in question is found by $\texttt{pred}(\texttt{node\_rank}_{\texttt{POST}}(v_{k_1}))$ on the data structure containing the nodes at level $\texttt{depth}(v)$. As the number of selected nodes in total is $O(n/\Delta)$, the total space cost of all the predecessor structures is $O((n/\Delta)\cdot\lg n)$ bits. The time complexity is $O(\lg\lg n)$.

Combining these numbers we obtain the number of beer nodes in the given rectangle.

Thus if we store P explicitly, then the space required is $O((n/\Delta)\cdot\lg n)+n+o(n)$ with time $O(\lg\lg n)$.

If we do not store P explicitly, then the space required is $O((n/\Delta)\cdot\lg n)+o(n)$ with time $O(\lg n)$. $\square$

Type 3 Rectangles: Type 3 rectangles are similar to type 2 rectangles. As above, we use the same rectangle $(u,v)\times(\texttt{node\_rank}_{\texttt{POST}}(v),\texttt{node\_rank}_{\texttt{POST}}(u))$ for the selected nodes. We again use the rectangle $[\texttt{depth}(u'),\texttt{depth}(v'))\times(\texttt{node\_rank}_{\texttt{POST}}(v_{\texttt{depth}(u')}),\texttt{node\_rank}_{\texttt{POST}}(u')]$ to capture the complete subtrees that we wish to use. The incomplete subtree is exactly the same as in type 2, so we are able to apply Lemma 5.3.7.

Thus putting everything together we have the following theorem:

Theorem 5.3.8. *Let G be a beer proper interval graph. Fix Δ, then G can be represented using $3n+o(n)+O((n/\Delta)\cdot\lg n)$ bits and can support* `adjacent`, `degree`, `neighborhood`, `distance` *in $O(1)$ time,* `spath`, `beer_spath` *in $O(1)$ time per vertex on the path and* `beer_distance` *in $O(\Delta+\lg n)$ time.*

In particular, if we take $\Delta=\lg n$, then the space is $O(n)$ with time $O(\lg n)$ and if we take $\Delta=f(n)\lg n$ for some $f(n)=\omega(1)$, then the space is $3n+o(n)$ and the time is $O(f(n)\lg n)$.

If we wish to further our trade off and improve the time, we must explicitly store P as in the proof Lemma 5.3.7, and use the space inefficient (but time efficient) range query data structures. Thus we obtain:

Theorem 5.3.9. *Let G be a beer proper interval graph. Fix Δ, then G can be represented using $4n+o(n)+O((n/\Delta)\cdot\lg n\lg\lg n)$ bits and can support* `adjacent`, `degree`, `neighborhood`, `distance` *in $O(1)$ time,* `spath`, `beer_spath` *in $O(1)$ time per vertex on the path and* `beer_distance` *in $O(\Delta+\lg\lg n)$ time. In particular, if we take $\Delta=\lg\lg n$, then the space is $O(n\lg n)$ with time $O(\lg\lg n)$.*

5.4 Interval Graphs

In this section, we study how to compute and construct data structures for beer paths and beer distances. We will begin with Theorem 4.4.3, which stores the distance tree T and the navigational data structure of Acan et al. [2]. The major difference between interval graphs and proper interval graphs is how adjacency can be checked. In a proper interval graph, for a vertex v, and its parent p in the distance tree, v is adjacent to every vertex between v and p by Lemma 4.4.6. However, in interval graphs, this is not the case, and depending on the graph structure, any of those vertices can be adjacent or not adjacent to v.

5.4.1 Calculating Beer Distance

As in proper interval graphs, we begin by investigating the conditions in which nodes *preserve* the distance.

Definition 5.4.1. Given two vertices u, v, for a node w (such that $u < w < v$), we say that w is ***+k the distance*** (w.r.t. u, v) if $\texttt{distance}(u, v) + k = \texttt{distance}(u, w) + \texttt{distance}(v, w)$.

We note that preserving the distance is equivalent to being +0 the distance. So, using w that is $+k$ the distance as a beer node will add k to the optimal (non-beer path) distance. To do this, we will add one more condition to that of Lemma 5.3.3.

Let u be a node in T. As shown in the proof of Lemma 5.3.6, the node on the next level that splits it in the same way as u is the largest neighbour of u. We denote this by $\texttt{last}(u)$. For example, in Example 5.3.1, the largest neighbour of the node 8, is the node 13, as 8 is adjacent to node 13, but not node 14. Thus $\texttt{last}(8) = 13$.

Lemma 5.4.2. *Let $u < v$ be vertices in a beer interval graph G with depths $\texttt{depth}(u) = k_1 \leq k_2 = \texttt{depth}(v)$. Consider the nodes $u < w < v$. Then we have the following two criteria:*

- *If* $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(v)$, *then either*

  $$\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v) \textit{ or } \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$$

  If $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) > \texttt{node\_rank}_{\texttt{POST}}(v)$, *then*

  $$\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$$

- *If* $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(v)$, *then*

  $$\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) > \texttt{node\_rank}_{\texttt{POST}}(v) \textit{ or}$$

$$\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) < \texttt{node\_rank}_{\texttt{POST}}(w)$$

If $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v)$, *then*

$$\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) < \texttt{node\_rank}_{\texttt{POST}}(w)$$

If w satisfies both criteria, then w preserves the distance. If w satisfies one of the criteria, then w is +1 the distance and if w satisfies neither criteria, then w is +2 the distance.

We note that the 2 criteria encode the following: first is the same criteria as Lemma 5.3.3 which check if w would preserve the distance. The second criteria checks that we may reach w from v_k (i.e. one of the ancestors of v) in one step (i.e. w is adjacent to v_k) or not. Thus clearly if both are true, then we are in the same situation as Lemma 5.3.3, while each failed condition add 1 to the distance.

Proof. Consider the path to the root from v: $v_1, \ldots, v_{k_2} = v$, and let k be the node where $v_k \leq \texttt{last}(u) < v_{k+1}$. As in Lemma 5.3.3, let the nodes $V_i = \{u < w < v; v_i \leq w < v_{i+1}\}$, which we will call slices.

We wish to split V_i based on the nodes distances to u. We will show that $V_i = V_i^+ \cup V_i^-$ where $w \in V_i^+$ if $\texttt{distance}(w, u) = \texttt{distance}(v_i, u)$ and $w \in V_i^-$ if $\texttt{distance}(w, u) = \texttt{distance}(v_{i+1}, u)$.

By the distance algorithm, suppose that $x \in V_i^+$, then any children of x, c, is in V_{i+1}^+. We can see this as $\texttt{distance}(c, u) = \texttt{distance}(x, u) + 1 = \texttt{distance}(v_i, u) + 1 = \texttt{distance}(v_{i+1}, u)$.

Suppose that $\texttt{depth}(\texttt{last}(u)) = k$, so that $\texttt{last}(u)$ is on the same level as v_k. Then $V_i^+ = \{w \in V_i; \texttt{depth}(w) = i, \texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$ and V_i^- is the remaining nodes. Otherwise, if $\texttt{last}(u)$ is on the same level a v_{k+1}, then $V_i^+ = \{w \in V_i; \texttt{depth}(w) = i \text{ or } \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$.

We show this by induction. Consider the first slice that is non-empty, that is V_k. In the first case that $\texttt{depth}(\texttt{last}(u)) = k$, the nodes in $\{w \in V_k; \texttt{depth}(w) = k, \texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$, are those adjacent to u, and everything else is non-adjacent, but are adjacent to v_k, hence all other nodes have a distance of 2.

On the other hand, if $\texttt{depth}(\texttt{last}(u)) = k + 1$, then the nodes $\{w \in V_k; \texttt{depth}(w) = k \text{ or } \texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$ are those that are adjacent to u and have distance 1, and the rest have distance 2.

Since if a node x satisfies $\texttt{node\_rank}_{\texttt{POST}}(x) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$ if and only if any children of x, c also satisfies $\texttt{node\_rank}_{\texttt{POST}}(c) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$, we have that

$V_i^+ = \{w \in V_i; \texttt{depth}(w) = i, \texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$ or $V_i^+ = \{w \in V_i; \texttt{depth}(w) = i$ or $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))\}$.

For a node w, consider the interval $(w, \texttt{last}(w)]$. We are interested in whether this interval contains one of the nodes v_i. We use the open interval on the left because if $w = v_k$ for some k, then $\texttt{last}(w) \geq v_{k+1}$, so it still contains one of the v_i's, and by doing so, we make sure that exactly one v_i can be contained in the interval.

First suppose that $(w, \texttt{last}(w)]$ contains one such node, say v_{k-1}. Then $w \in V_k$ and w is adjacent to v_{k+1}. Thus $\texttt{distance}(v_k, v) = \texttt{distance}(w, v)$ by the distance algorithm. Furthermore, if $w \in V_k^+$, then $\texttt{distance}(u, w) = \texttt{distance}(v_k, u)$, so that $\texttt{distance}(u, v) = \texttt{distance}(u, w) + \texttt{distance}(v, w)$. On the other hand, if $w \in V_k^-$ instead, then $\texttt{distance}(u, v) + 1 = \texttt{distance}(u, w) + \texttt{distance}(v, w)$.

Next suppose that $(w, \texttt{last}(w))$ does not contain any v_i, and thus, for some k, $v_k < w < v_{k+1}$. Since w is not adjacent to v_{k+1}, then we have $\texttt{distance}(w, v) = \texttt{distance}(v_k, v) + 1$. Therefore, if $w \in V_k^+$, then $\texttt{distance}(u, v) + 1 = \texttt{distance}(u, w) + \texttt{distance}(v, w)$ and if $w \in V_k^-$, then $\texttt{distance}(u, v) + 2 = \texttt{distance}(u, w) + \texttt{distance}(v, w)$.

Finally we wish to write the criteria that the interval $(w, \texttt{last}(w)]$ contains one of the v_i in a more computable form. In the first case that $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(v)$, so that w is to the left of the path, we need that either $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) > \texttt{node\_rank}_{\texttt{POST}}(v)$ or $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) < \texttt{node\_rank}_{\texttt{POST}}(w)$. The first case capture when the interval stays on the same level in the tree, and the second captures when the interval wraps to the next. In the second case that $\texttt{node\_rank}_{\texttt{POST}}(w) \geq \texttt{node\_rank}_{\texttt{POST}}(v)$, we need that $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(w)) < \texttt{node\_rank}_{\texttt{POST}}(w)$. $\square$

Again, we will find several sets that cover the beer nodes, and argue about the optimal beer node in these sets. We then take the minimum distance of these candidates. As before, we will assume that neither u nor v are beer nodes.

Candidate 1: This is the same as the proper interval graphs: $\{w \in B; w > v\} = \{w \in B; l_w > l_v\}$. We again claim that the best beer node is the smallest one. It turns out the exact same proof of Lemma 5.3.4 will work here.

Candidate 2: We wish to use the symmetric set of Candidate 1. Unfortunately, in interval graphs, this is not as simple. The set is $\{w \in B; r_w < r_u\}$. We note that this condition and the proper interval graph non-nesting condition gives $l_w < l_u$ so that $w < u$, and thus this is the right analogous set to consider. We claim that the best node is the node with the largest r_w in this set. This can be seen by reflecting the intervals of the vertices - equivalent to sorting them by the right endpoints instead. In this reflected graph, apply Lemma 5.3.4, and the result follows.

Candidate 3: The nodes $\{w \in B; u < w < v\}$. By Lemma 5.4.2, we can obtain the distance of the best beer node using the criteria in the lemma.

Candidate 4: The left over nodes. The nodes that do not belong to the previous candidate sets are w with: $l_w < l_v$ and $r_w > r_u$ and $l_w < l_u$. Thus these are the nodes with $l_w < l_u < r_w$, so they are adjacent to u. Formally, this is the set: $\{w \in B; w < u, r_w > r_u\}$. As these nodes are adjacent to u, all we need to check is their distance to v.

First consider the path from u to v. As in Lemma 4.4.2, we have the path to the root from v: $v = v_{k_2}, \ldots, v_0 = r$, And suppose that k is the index such that $v_k \leq u < v_{k+1}$. The end of the path could look like either $u, v_{k+1}, \ldots$ or $u, v_k, v_{k+1}, \ldots$, depending on whether v_{k+1} is adjacent to u or not (i.e. $l_{v_{k+1}} \leq r_u$?).

First assume that v_{k+1} is not adjacent to u, then for any possible candidate w, if w were adjacent to v_{k+1} (that is $\texttt{last}(w) \geq v_{k+1}$), then w preserves the distance (as $\texttt{distance}(w, v) = \texttt{distance}(u, v) - 1$). Otherwise, as $r_w > r_u > l_{v_k}$, w is adjacent to v_k and hence $\texttt{distance}(w, v) = \texttt{distance}(u, v)$ and w is $+1$ the distance.

Next assume that v_{k+1} is adjacent to u. Then again for any candidate w, $l_w < l_u < l_{v_k+1} < r_u < r_w$, so w is adjacent to v_{k+1} and w is $+1$ the distance. We note that w cannot be adjacent to v_{k+2} as in this case we would contradict that fact that v_{k+1} is the smallest node adjacent to v_{k+2}, by the parent relationship in T.

Finally we note that the only property of w that we used is that w is adjacent to u, and that if $x > u$ is adjacent to u, then w is also adjacent to x and hence we may relax the set to $\{w \in B; w < u, \texttt{last}(w) \geq \texttt{last}(u)\}$.

5.4.2 Data Structure for Beer Distance

We discuss how to use the previous results to create a data structure for the queries. We begin with the data structure derived in Theorem 4.4.3, which supports the interval graph queries in optimal time. This uses $n \lg n + O(n)$ bits of space. We store a bit vector B as before, which stores which nodes are beer nodes in level-order. This take $n + o(n)$ bits.

Candidate 1, we handle this in exactly the same way as in proper interval graphs, by using `rank` and `select` queries on B.

Candidate 2, we need to be able to find nodes in the mirrored graph.

Lemma 5.4.3. *We can find the desired node in the mirrored graph using $n \lg n + O(n)$ bits in $O(1)$ time.*

Proof. To find the appropriate node, we need to be able to implement the following steps:

1) For a node u, what is its index in the mirrored graph?

2) For a node u in the mirrored graph, what is the smallest beer node larger than it?

3) For a node u in the mirrored graph, what is its index in the original graph?

For step 1, for a node u, we get its interval right endpoint r_u. In the data structure of Acan et al. explained when we abstracted it to Theorem 4.3.7, we have a length $2n$ bit vector, which stores whether the endpoint at any index i is a right endpoint or a left endpoint. We use the rank operation to find how many intervals i have right endpoints less than r_u, and thus u is the $n - i$th node in the mirrored graph.

For step 2, we store the analogous bitvector B_R for the mirrored graph, which says whether vertex i in the mirrored graph is a beer node, and use it to find the appropriate beer node. This takes $n + o(n)$ bits.

For step 3, We would like to be able to map the vertex i in the mirrored graph (which corresponds to the vertex with the $n - i$-th largest right end point) to the corresponding vertex in the original graph. That is we wish to find the vertex v such that the right endpoint of v is the $n - i$-th largest right end point. As we stated in Theorem 4.3.7, there are 3 obvious ways of implementing D the data structure used to report right endpoints of intervals. One such way is storing a permutation P such that the right endpoint of v is the $P[v]$-th largest right endpoint. If we store the interval graph with D being a permutation Lemma 2.3.8, then this operation is exactly $P^{-1}(n - i)$.

In total, this takes $O(n)$ bits, and all of these operations are $O(1)$ except P^{-1} which using the parameter of Lemma 2.3.8 uses $O(f(n))$ time. □

Candidate 3 We are able to handle this using 3D 5-sided orthogonal range emptiness data structures.

Lemma 5.4.4. *We can check if a beer node satisfies the criteria of Lemma 5.4.2 using a constant number of 3D 5-sided orthogonal range emptiness data structures. Thus the space/time requirements are either $O(n \lg n)$ space and $\lg^{\varepsilon} n$ time or $O(n \lg n \lg \lg n)$ space and $\lg \lg n$ time.*

Proof. We will again use the range emptiness method described for proper interval graphs. However, as we need an additional criterion, we also need another dimension in the grid to store this criterion. Thus for each beer node b, we store the following point in a 3D table: $(b, \texttt{node\_rank}_{\texttt{POST}}(b), \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(b)))$.

We note that it is difficult to express the condition

$$\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(b)) < \texttt{node\_rank}_{\texttt{POST}}(b)$$

as it depends on the values of the node being filtered. Furthermore in the second criterion, we would need 6-sided rectangles. To alleviate this, we create 2 tables: 1 for beer nodes whose intervals stay on the same level (that is $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(b)) > \texttt{node\_rank}_{\texttt{POST}}(b)$) and one for those that wrap to the next level. We will names these R_1 and R_2.

To do this, we will first query the criteria separately, and then query them together. If either of the criterion returns a positive, then we know that the optimal beer node is either $+0$ or $+1$ the distance, depending on the result of the joint query. If neither the criteria return a candidate, then the optimal node is $+2$ the distance (we will need to check using B that there is a beer node in this range to use).

For the first criterion, directly translating the condition from Lemma 5.4.2, we obtain the following rectangles.

- $(u, v) \times (-\infty, \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))) \times [-\infty, \infty]$,
- $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \infty) \times [-\infty, \infty]$, and
- $(u, v) \times (\texttt{node\_rank}_{\texttt{POST}}(v), \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))) \times [-\infty, \infty]$.

As we have split the beer nodes into two data structures R_1 and R_2, we need to do the query on both.

To check only the second criterion, we have the following rectangles:

Does $(u, v) \times [-\infty, \texttt{node\_rank}_{\texttt{POST}}(v)] \times [\texttt{node\_rank}_{\texttt{POST}}(v), \infty]$ in R_1 or $(u, v) \times [-\infty, \texttt{node\_rank}_{\texttt{POST}}(v)] \times [-\infty, \infty]$ in R_2 contain any nodes?

Does $(u, v) \times [\texttt{node\_rank}_{\texttt{POST}}(v), \infty] \times [\texttt{node\_rank}_{\texttt{POST}}(v), \infty]$ in R_2 contain any nodes?

Finally to check both criteria at the same time, we take the intersection of the rectangles from the two separate criteria. As the intersection of rectangles are rectangles, with potentially more sides, we may do this. Since the third coordinate is always $[-\infty, \infty]$ in criterion 1, and it is open ended on at least 1 side in criterion 2, we see that any intersection is at most a 5 sided rectangle. □

Candidate 4 We will use the following lemma:

Lemma 5.4.5. *We can convert the criterion of candidate 4 into a constant number of 5-sided rectangles.*

Proof. The nodes we are interested in are those with $w < u$ and $\texttt{last}(w) \geq \texttt{last}(u)$. Let $p(u)$ denote the parent of u in T. All such w are adjacent to $\texttt{last}(u)$ and thus $p(\texttt{last}(u)) \leq w < u$. First we find whether this Candidate set is empty or not. As we are checking the condition $\texttt{last}(u) \in [w, \texttt{last}(w)]$, this will be similar to the second criterion of candidate 3. The rectangles are:

- $[p(\texttt{last}(u)), u) \times [-\infty, \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))] \times [\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)), \infty]$ in R_1,
- $[p(\texttt{last}(u)), u) \times [-\infty, \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))] \times [-\infty, \infty]$ in R_2,
- $[p(\texttt{last}(u)), u) \times [\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)), \infty] \times [\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)), \infty]$ in R_2.

As described in the previous part, there are two cases, either v_k is adjacent to u or v_k is not adjacent to u (we check this in $O(1)$ time from the distance algorithm).

In the case that v_k is not adjacent to u. We wish to find a node $w < u$ such that $\texttt{last}(w) > v_k$ (so that $v_k \in [w, \texttt{last}(w)]$). Any such w is adjacent to v_k and must satisfy $p(v_k) \leq w < v_k$. Thus we replace all instances of $\texttt{last}(u)$ with v_k in the rectangles above. If a node exists then it preserves the distance, and if no such node is found, then the best possible is +1 the distance.

In the case that v_k is adjacent to u, we do not need to do anything more, since any w is +1 the distance. ☐

Finally, to handle shortest paths, we use the reporting query rather than the emptiness query. When the reporting query returns the first point, we stop. After we find the best beer node, we list out the path using two `spath` queries.

Theorem 5.4.6. *Let G be a beer interval graph, with beer nodes B. The there exists a data structures using $n \lg n + o(n \lg n) + O(|B| \lg n)$ bits that supports* `degree`, `adjacent`, `distance` *in $O(1)$ time,* `neighborhood`, `spath` *in $O(1)$ time per vertex in the path/neighbourhood,* `beer_distance` *in $O(\lg^\epsilon n)$ time and* `beer_spath` *in $O(\lg^\epsilon n + d)$ time where d is the distance between the two vertices.*

Alternatively, we may increase the space from $O(|B| \lg n)$ to $O(|B| \lg n \lg\lg n)$ and replace the $\lg^\epsilon n$ in `beer_spath`, `beer_distance` *with $\lg\lg n$.*

Proof. We store the interval graph using Theorem 4.4.4, where D is a permutation stored using Lemma 2.3.8, where we set the parameter $f(n) = \lg\lg n$.

Other extra space needed is the range search data structures occupying $O(|B| \lg n)$ bits and various bit vectors occupying $O(n)$ bits.

The query time for `degree`, `adjacent`, `distance`, `neighborhood`, `spath` are inherited from Theorem 4.4.4. `beer_distance` requires the computation of P^{-1} and range emptiness queries, using either $O(\lg^\epsilon n + \lg\lg n) = O(\lg^\epsilon n)$ time or $O(\lg\lg n)$ time. `beer_spath` requires $O($`beer_distance`$)$ time to find the candidate, followed by $O(d)$ time to list out the vertices on the path. □

5.5 Lower Bound

Although we have given upper bounds in the forms of data structure for both beer interval graphs and beer proper interval graphs, we have not shown that the data structures are succinct. In this section, we will prove lower bounds for both beer proper interval graphs and beer interval graphs by enumerating them.

Our first task is to formalize exactly what we are enumerating. Just as when we are enumerating graphs, we need to consider isomorphisms, we when enumerate beer graphs, we must also consider isomorphisms. Thus we must formalize what it means for two beer graphs to be isomorphic.

5.5.1 Beer Graph Isomorphisms

By definition, a beer graph (G, B) is a graph G together with a set $B \subseteq V$ of beer vertices. We will refer to B as a beer vertex pattern. We will say that two beer graphs (G_1, B_1) and (G_2, B_2) are isomorphic (and thus are the same object) if there exists a bijection $f : V(G_1) \mapsto V(G_2)$ such that $(u, v) \in E(G_1) \Leftrightarrow (f(u), f(v)) \in E(G_2)$ and $u \in B_1 \Leftrightarrow f(u) \in B_2$. The first condition is the standard condition for two graphs to be isomorphic and the second condition says that this isomorphism also preserves beer vertices. This is the natural definition as under this isomorphism, any path P between two vertices u, v is a beer path if and only if the corresponding path $f(P)$ between $f(u), f(v)$ is a beer path, and thus the isomorphism preserve beer distances. Under this definition, for two beer graphs to be isomorphic, the underlying graphs must also be isomorphic as well.

It may be tempting to think that we have a subset of beer vertices B out of set V of size n. Therefore, we must need n bits to represent this subset. Our main motivating example shows that this is not the case.

Example 5.5.1. Suppose our graph class are cliques, then how many beer cliques are there? On n vertices, there is exactly one underlying graph $G = K_n$ on n vertices that is a clique. Thus it remains to see how many different beer vertex patterns we can have. By definition, if (K_n, B_1) were isomorphic to (K_n, B_2), then there exists an automorphism σ of K_n mapping vertices $u \in B_1$ to $\sigma(u) \in B_2$ bijectively, and thus $|B_1| = |B_2|$. Conversely, if $|B_1| = |B_2|$ then there exists a bijection σ that maps the elements of B_1 to B_2 and fixes every other vertex. As the underlying graph is the complete graph K_n, this σ is also an automorphism of the underlying graph as well. Thus (K_n, B_1) is isomorphic to (K_n, B_2) exactly when $|B_1| = |B_2|$. The number of different ways to add beer nodes to a clique on n vertices is thus $n + 1$. $\square$

As $(G_1, B_1) \cong (G_2, B_2)$ happens only when $G_1 \cong G_2$, it remains to develop the theory to compute the number of beer vertex patterns that are different when given a specific underlying graph G (and then to find the number of beer graphs, sum over all graphs that we are considering). Let $Aut(G)$ denote the automorphism group of a graph G. We will view $B \subseteq V$ as a vector $B \subseteq 2^n$ on the hypercube (where the i-th bit denotes whether the i-th vertex belong to the set or not), and $Aut(G)$ as a group that acts on 2^n. In this lens, two beer vertex patterns B_1, B_2 are the same if there exists a group element σ mapping B_1 to B_2, and thus B_1 and B_2 belong to the same orbit of this group action. The number of different beer vertex patterns is thus the number of orbits $|2^n/Aut(G)|$. To count the number of orbits, we will use the Polya enumeration theorem [71], which in its most basic form, states that if we denote $c(\sigma)$ as the number of cycles in σ when viewed as a permutation of $V(G)$, $|2^n/Aut(G)| = \frac{1}{|Aut(G)|}\sum_{\sigma \in Aut(G)} 2^{c(\sigma)}$.

Now let us apply this to arbitrary beer graphs. Without even considering the automorphism groups of the graphs, we can see that the minimal automorphism group is the one element group (i.e. the graph has no automorphisms), and the largest automorphism group is S_n corresponding to a clique. In the first case, $\frac{1}{|Aut(G)|}\sum_{\sigma \in Aut(G)} 2^{c(\sigma)} = 2^n$ and the second $\frac{1}{|Aut(G)|}\sum_{\sigma \in Aut(G)} 2^{c(\sigma)} = n + 1$.

Therefore, if X is a class of graphs, then the number of beer graphs of this class is between $(n + 1)|X|$ and $2^n|X|$. Applying this to beer interval graphs (with $\lg|X| = n\lg n - o(n\lg n)$) and we see that the lower bound for beer interval graphs is between $n\lg n + \lg n$ and $n\lg n + n$. In both bounds the term contributed by the beer vertices is a lower order term. Thus the lower bound for beer interval graphs (and any other class of graphs with a lower bound $\omega(n)$) remains the same at $n \lg n$ bits. This also shows that our data structure from Theorem 5.4.6 is not succinct for $|B| = \Theta(n)$, but rather it is compact.

Next we will consider beer proper interval graphs. Our upper bound data structure of Theorem 5.3.8 uses $3n + o(n)$ bits, while a lower bound inherited from proper interval graphs is $2n - o(n)$ bits. Thus the analysis above shows that the true lower bound is αn bits for some $2 \leq \alpha \leq 3$. If $\alpha = 3$, then our data structure is succinct. However, we will ultimately show in this section that the lower bound is $\lg(4 + 2\sqrt{3})n \approx 2.9n$ bits, so that there is more work to be done in creating a succinct data structure. To compute this, we will need a better understanding of the automorphism groups of proper interval graphs.

5.5.2 Automorphism Groups of Proper Interval Graphs

Klavic and Zeman [52] showed that $Aut(\text{connected PROPER INT}) = Aut(\text{CATERPILLAR})$. A caterpillar graph/tree is a path together with a set of leaves that are adjacent some vertex on the path. In particular, the automorphism group of any particular connected proper interval graph is generated by 2 types of automorphisms. First are automorphisms that swap twin vertices - which corresponds to those that swap the leaves adjacent to the same vertex on the path of a caterpillar graph. In a proper interval graph, twin vertices are those that have the same set of maximal cliques. The second is an automorphism that reverses the proper interval graph, which corresponds to reversing the path of a caterpillar graph. This reversal corresponds to a reversal of the maximal cliques. Of course, for any particular graph, there may not be any twin vertices, and thus there are no automorphisms of the first type. As for the second type, it can only exist when the number of vertices in the maximal cliques are symmetrical - as the vertices in the first maximal clique are mapped to those in the last maximal clique etc.

We will assume that the maximal cliques are not symmetrical and thus no automorphisms of the second type exists. To see this, we may always desymmetrize the sequence of maximal cliques by adding one vertex to only the first maximal clique if necessary.

Now suppose that G is a connected proper interval graph. As being twin vertices are an equivalence relation, let $S_1, \ldots, S_h$ be the equivalence classes of twin vertices, that is $u, v \in S_i$ implies that u, v are twins. Let $k_i = |S_i|$ and we will say that vertices which have no twins are in a class of size 1, so that $\sum_i k_i = n$.

Lemma 5.5.2. *Let G be a proper interval graph with twin vertex classes of sizes $|S_1| = k_1, \ldots, |S_h| = k_h$. Then $|2^n / Aut(G)| \leq (k_1 + 1)(k_2 + 1) \cdots (k_h + 1)$. This is an equality in the case that the graph is connected and the maximal cliques are non-symmetrical.*

Proof. In the case that the graph is connected and the maximal cliques are non-symmetrical, we have only type 1 automorphisms.

Let σ_i be a permutation that permutes only those vertices of S_i and σ_j permuting those of S_j, then $\sigma_i\sigma_j = \sigma_j\sigma_i$ as $S_i \cap S_j = \emptyset$. Thus we may write $Aut(G) \cong \mathbb{S}_{k_1} \times \cdots \times \mathbb{S}_{k_h}$ where $\mathbb{S}_n$ denotes the symmetric group (set of all permutations) on n elements.

Applying this to Polya enumeration theorem, we obtain that

$$\begin{aligned}
|2^n/Aut(G)| &= \frac{1}{|Aut(G)|} \sum_{\sigma \in Aut(G)} 2^{c(\sigma)} \\
&= \frac{1}{k_1! \cdot k_2! \cdots k_h!} \sum_{\sigma_1 \times \sigma_2 \times \cdots \sigma_h \in \mathbb{S}_{k_1} \times \cdots \times \mathbb{S}_{k_h}} 2^{c(\sigma_1 \times \sigma_2 \times \cdots \sigma_h)} \\
&= \left(\frac{1}{k_1!} \sum_{\sigma_1 \in \mathbb{S}_{k_1}} 2^{c(\sigma_1)} \right) \left(\frac{1}{k_2!} \sum_{\sigma_2 \in \mathbb{S}_{k_2}} 2^{c(\sigma_2)} \right) \cdots \left(\frac{1}{k_h!} \sum_{\sigma_h \in \mathbb{S}_{k_h}} 2^{c(\sigma_h)} \right) \\
&= |2^{k_1}/Aut(K_{k_1})| \cdots |2^{k_h}/Aut(K_{k_h})| \\
&= (k_1 + 1)(k_2 + 1) \cdots (k_h + 1)
\end{aligned}$$

The last equality comes from our Example 5.5.1 dealing with cliques.

Finally, we note that by the definition of group action, if $H_1 \subset H_2$ are two groups acting on a set X, then the orbit of any element $x \in X$ under H_2, $H_2 \cdot x = \{h \cdot x; h \in H_2\}$ is a superset of that of the orbit under H_1. Thus the number of orbits $|X/H_2| \leq |X/H_1|$. As the above equation applies exactly to non-symmetric connected proper interval graphs, and dropping the connectedness/symmetric property only increases the automorphism groups (if two connected components are isomorphic, then there is an automorphism that swaps the two connected components), we conclude that if G were a proper interval graph instead, we may say that $|2^n/Aut(G)| \leq \Pi_i(k_i + 1)$.

□

As a further remark, we see that any vertex that is not a twin contributes a multiplicative factor of 2 to the above quantity (intuitively, this means that you must store a bit stating whether this vertex is a beer vertex or not) while any vertex that have twin vertices contributes a much smaller term (intuitively, this means that it is necessary to only store only the number of beer vertices using $\lg k_i$ bits, furthering our intuition from Example 5.5.1).

For a proper interval graph, we will also refer to the above quantity as its weight, as the number of beer proper interval graphs would be the weighted sum of proper interval graphs.

We will first apply the above using a method of bounding the number of beer proper interval graphs. We will then use it to compute a recurrence which allows us to compute the lower bound exactly. It is our hope that the techniques used in bounding the number of beer proper interval graphs will translate to a method of constructing a data structure using strictly less than $3n$ bits of space, whereas the recurrence seems less useful in that regard.

5.5.3 Representation of Proper Interval Graphs

A proper interval graph can be represented using a length $2n$ bitvector, where the i-th vertex's left and right end point are the indices of the i-th 1 and 0 respectively. Conversely, given a length $2n$ bitvector such that at any index, the number of 1s in the prefix is at least the number of 0s (i.e. $\texttt{rank}_1(i) \geq \texttt{rank}_0(i)$, so that every 0 - the right end point of an interval has a matching 1 - the left end point of an interval) we may view it as the bitvector of some proper interval graph.

A maximal clique is a clique that is maximal under subsets. An equivalent characterization of Interval graphs is that the set of maximal cliques can be linearly ordered so that for any vertex v, the set of maximal cliques containing v is contiguous in the ordering [34]. Furthermore if graph is a connected proper interval graph then this order is unique up to reflection [53] (as connected proper interval graphs by definition of no nested intervals are overlap connected). Thus, the mapping $B \mapsto G$ sending a bitvector with the above property to a proper interval graph G is an onto map, and furthermore if we restrict this to only generate connected graphs, no graph is generated more than twice.

To obtain an ordering of the maximal cliques from the bitvector, we consider the 1s and 0s as blocks. The end points of the i-th vertex is now the block number that the i-th 1 or 0 belongs to.

We will refer to the intervals generated by the bitvector indices as the bitvector representation of the graph and by the intervals generated from the block representation as the clique representation of the graph.

Lemma 5.5.3. *Given a proper interval graph, and its interval representation using a length $2n$ bit vector, there are k maximal cliques where k is the number of blocks of 1s (and 0s) in the bit vector. If we view the i-th vertices' left and right endpoints by the block that the i-th*

1 and 0 belongs to, then i-th maximal clique contains all vertices whose intervals contain i.

Proof. To see that this indeed gives us the desired maximal cliques, consider any maximal clique C. Then the intervals of the vertices $v \in C$ pairwise intersect and thus there exists some number l that belongs to all the intervals. Conversely all intervals containing l forms a clique and must be equal to C. Thus all maximal cliques are found by taking some number l and taking all intervals containing l.

Let l be the index of the last 1 in some block of 1s in the bitvector. We show that l gives a maximal clique. Let C_l be the set of vertices whose intervals contain l, and suppose that there exists another number $l' > l$ (other case is symmetrical) whose clique $C_{l'}$ strictly contains C_l. Let $v \in C_{l'} \setminus C_l$. The left endpoint of v is to the right of l. Since l is the index of the last 1 in a block of 1s, the left endpoint of v must be to the right of the block of 0s immediately following l, and thus so much l'. But this block of 0s represents the right endpoint of vertices of C_l, which then cannot contain l'.

By collapsing the 1s and 0s into blocks, the vertices whose intervals contain i (in the block view) are exactly those whose intervals (in the bitvector view) contain l_i = the index of the last 1 in the i-th block of 1s.

Conversely, consider any index i such that it is not the last 1 of a block. There are 3 cases, i is between two numbers in the bitvector of the form $11, 00, 01$. In the first case and third cases, moving i to the right past the next 1 increases the clique (that 1 represents a new vertex whose interval now contains i). In the second case, moving i to the left past the 0 increases the clique (that 0 represents the right endpoint of some vertex that now contains i). □

Example 5.5.4. The proper interval graph represented by the following bit sequence 1101011000 has maximal cliques $\{1, 2\}, \{2, 3\}, \{3, 4, 5\}$ and in this arrangement, the cliques that contain any vertex are consecutive.

It is obvious each length $2n$ bitvector that encodes a proper interval graphs must be balanced (at any index, the number of 1 preceding must be at least the number of 0s preceding) so that each interval's right coordinate is larger than its left coordinate, and thus can be viewed as a Dyck path or a balanced parenthesis sequence. Consider any index where the Dyck path touches the line $x = 0$. Any vertex represented by an interval to the left of this point does not intersect interval of a vertex to the right of this point, and thus the graph is disconnected. A proper interval graph is connected then if the Dyck path never touches the line $x = 0$, except at the two end points. Thus connected proper interval graphs correspond to irreducible dyck paths.

Let k be the number of maximal cliques (and the number of blocks of 1s and 0s in the bitvector representation). Then the sizes of the blocks of 1s forms a composition of n. We will represent this by a set of $k - 1$ barriers that split the n nodes into k parts. There are $n - 1$ possible positions that could have barriers (between positions i and $i + 1$ for $i = 1, \ldots n - 1$). The positions that do not have barriers will be denoted by the set R_l (l for the left end point of intervals). Similarly, the sizes of the blocks of 0s is also a composition of n and can be represented by a set of $k - 1$ barriers. The positions which do not have barriers will be denoted by R_r (r for right end points). Finally, let $R_I = R_l \cap R_r$ be the intersection of the two.

Conversely, given two compositions of n, we can recover the block sizes and thus the bitvector.

Lemma 5.5.5. *Two sets R_l, R_r represents a proper interval graph if at any index i, the set $R_l(i) = \{x \in R_l; x < i\}$ is at least as large as the set $R_r(i) = \{x \in R_r; x < i\}$.*

Proof. Consider the clique representation of a proper interval graph. For any vertex i, we must have that the left end point is at most equal to the right end point. Translating to R_l, R_r, at any index i, the number of barriers preceding i in R_l is at most that of the number of barriers preceding i in R_r (the block number of the i-th 1/0 is equal to the number of barriers preceding $i + 1$ in R_l/R_r respectively). Thus the number of positions which do not have barriers in $R_l : R_l(i)$ must be at least $R_r(i)$. $\square$

Given two sets R_r, R_l, we say that they satisfy the Dyck path property if they represent a proper interval graph, and we will denote R_r, R_l as the barrier or composition representation of the graph.

Example 5.5.6. In our graph above whose bitvector representation was 1101011000, the sizes of the blocks of 1s were $2, 1, 2$. Thus R_l would be 0010100 (or as a set $\{1, 4\}$), where the 1 represents the barrier and the 0 represents the size of the composition.

R_r would be 0101000 (or as a set $\{3, 4\}$) as the sizes of the blocks of 0s are $1, 1, 3$.

Finally R_I would be 01010100 as only position 4 is shared among R_l, R_r.

5.5.4 First Lower Bound

We will derive a subset of proper connected interval graphs that a) there are a lot of graphs in the subset, so that the number of bits to represent them is large and b) the number of bits needed to store the beer vertices is large, which consequently means the number of twin vertices is small.

As our lower bound will be information theoretic, we will be taking the lg. The number of graphs will be exponential in n and thus any poly$(n, 1/n)$ factors will be lower order terms. For simplicity, we will ignore them. Furthermore, proving a bound on a graph of $n + c$ vertices for constant c will also introduce lower order terms, and thus we will for simplicity ignore any constant increase in graph size.

We will use the barrier representation of a proper interval graph, R_l, R_r, as their structure and in particular R_I allows us to compute the sizes of the equivalence classes of twin vertices nicely.

Lemma 5.5.7. *Two vertices i and $i+1$ are twins if and only if there is no barrier between them in R_I.*

Proof. Two vertices i, $i+1$ has no barrier between them in $R_I \Leftrightarrow i, i+1$ has no barrier between them in R_l and $R_r \Leftrightarrow i, i+1$'s left endpoints belong to the same block of 1s and their right endpoints belong to the same block 0s $\Leftrightarrow i, i+1$ have the same set of maximal cliques $\Leftrightarrow i, i+1$ are twins. □

Thus the composition defined by R_I is exactly the sizes of the equivalence classes of twin vertices.

Example 5.5.8. Again in our graph represented by the bitvector 1101011000, the only twin vertices are $4, 5$ since $R_I = 01010100$.

For a graph on n vertices, there are $n - 1$ locations to place barriers. To make the calculations cleaner, we will consider $n + 1$ vertices so that there are n locations to place barriers.

Lemma 5.5.9. *Let R_r, R_l satisfy the Dyck path property for proper interval graphs on $n+1$ vertices. Let $x = |R_I|$ and $y = |R_r \setminus R_I|$. Then the number of such R_r, R_l is*

$$\sum_{x=0}^{n} \sum_{y=0}^{(n-x)/2} \binom{n}{x} \binom{n-x}{2y} \binom{2y}{y} \frac{1}{y+1}$$

Proof. We first select the x positions of R_I. Each of $R_l \setminus R_I$, $R_r \setminus R_I$ have size y, so from the remaining $n - x$ positions we select $2y$ elements for their union. Finally out of the $2y$ elements, we decide which set each element belongs.

To satisfy the Dyck path property, the $2y$ elements distributed to R_l, R_r must satisfy the Dyck path property. The number of ways to do this is exactly the number of Dyck paths of length $2y$ which is the y-th Catalan number $\binom{2y}{y}/(y+1)$. $\square$

Let $S_{x,y} = \{R_l, R_r; |R_I| = x, |R_l| = |R_r| = x+y, R_l, R_r \text{satisfies the Dyck path property}\}$. Then we have shown that $|S_{x,y}| = \binom{n}{x}\binom{n-x}{2y}\binom{2y}{y}\frac{1}{y+1}$.

We also note that as an aside, we have proven (or rediscovered) the following Catalan number identity (C_k denoting the k-th Catalan number), which may be useful for other applications.

$$C_{n+1} = \sum_{x=0}^{n} \sum_{y=0}^{(n-x)/2} \binom{n}{x}\binom{n-x}{2y} C_y$$

Let G be any proper interval graph on $n+1$ vertices and let R_l, R_r be the Dyck path representation. Consider the following graph G' on $n+3$ vertices, whose Dyck path representation is $R'_l = 0R_l10$, $R'_r = 01R_r0$. Note that by removing a barrier in R'_l at the first index and not in R'_r we obtain the following property: at any index i, $|R'_l(i)|$ is strictly greater than that of $|R'_r(i)|$ and this translates to a Dyck path that never touches the x-axis. Therefore, G' is a connected proper interval graph. Furthermore $R'_I = 01R_I10$ so that the sizes of the twin vertex classes are preserved - we add two more of size 1.

Thus if k_i are the sizes of the twin vertex classes of G, then

$$|2^n/G'| = 4\Pi_i(k_i+1)$$

We will drop the factor of 4 as it contributes to a lower order term.

Lastly, we wish to investigate the number of beer vertex patterns given only the number of parts in the composition representing the equivalence classes of twin vertices, as that is what R_I gives us.

Let $g(R_l, R_r) = g(R_I)$ be the number of beer vertex patterns. That is if k_i are the sizes of the parts in composition defined by R_I, then $g(R_I) = \Pi_i(k_i+1)$.

Let $f(n,x)$ be the average over all compositions of n with $n-x$ parts of the number of beer vertex patterns. That is

$$f(n,x) = \sum_y \sum_{R_l, R_r \in S_{x,y}} g(R_l, R_r) / \sum_y |S_{x,y}|$$

Unfortunately, it is difficult to compute $f(n,x)$ exactly, so the best we can do is bound it.

Lemma 5.5.10. $2^n(1/2)^x \le f(n,x) \le 2^n(3/4)^x$*. Up to poly*$(n, 1/n)$ *factors.*

Proof. Each $g(R_l, R_r)$ computes the number of beer vertex patterns for a composition of n into $n-x$ parts where $x = |R_I|$ (this might be off by 1 but that only contributes a constant factor). Consider two parts of sizes $k_1 < k_2$. These two parts contributes a factor of $(k_1+1)(k_2+1)$. Now consider two parts of k_1+1, k_2-1. The factor is now $(k_1+2)(k_2)$. By doing this we have increased our term by $k_2 - k_1 - 1 \ge 0$. Thus if we rearrange our composition by evening out the sizes, we achieve a larger total. Conversely, if we concentrate all of the composition into one term, we obtain the smallest total.

Thus $g(R_l, R_r)$ is maximized when all parts are as equal as possible and $g(R_l, R_r)$ is minimized when all parts have size 1 except the last part which has $n-x+1$. As $f(n,x)$ is the average of all composition, it is bounded by the largest valued compositions and the smallest valued composition.

Thus the lower bound for $f(n,x)$ is the composition $(1, 1, \ldots, n-x+1)$ which has value approximately $2^{n-x}(n-x+2)$. Removing poly factors we obtain the desired term $2^n(1/2)^x$.

For the upper bound, consider $0 \le x \le n/2$. In this region the maximum valued composition is $(1, \ldots, 2)$ where there are $n-2x$ 1s and x 2s. This composition has a total value of $2^{n-2x}3^x = 2^n(3/4)^x$.

For $n/2 \le x \le 2n/3$, the maximum valued composition is $(2, 2, \ldots, 3, 3)$. To see that the maximum of $2^n(3/4)^x$ holds, we show that when x increases by 1, the total decreases by a factor of at least $3/4$. To see this in this region, when x increases, we replace 3 parts of 2 by 2 parts of 3. That is we replace a factor of $27 = 3^3$ by $16 = 4^2$. As $16/27 < 3/4$ this holds.

For larger x we replace $k+1$ copies of k by k copies of $k+1$. The values are $(k+1)^{(k+1)}$ and $(k+2)^k$ which in all cases decrease the total by at least $(3/4)$. □

We are now finally ready to prove our first lower bound

Theorem 5.5.11. *To represent a beer proper interval graph G which is able to support* `adjacent` *and* `beer_distance` *will require at least* $(\lg 7)n - o(n) \approx 2.81n$ *bits in the worst case.*

Furthermore, the lower bound cannot be greater than $n \lg 15/2 \approx 2.91n$ *bits.*

Proof. Our condition on requiring `adjacent` and `beer_distance` is so that the beer graph (G, B) can be recovered. G is recovered from `adjacent` and $B[i]$ can be recovered using `beer_distance`(i, i).

First we consider the lower bound. For each beer proper interval graph G on $n+1$ vertices, we apply the transformation to make it connected on $n+3$ vertices. Since we are now only generating connected proper interval graphs, we do not generate each graph more than twice. We will also use the approximation 4^y to the y-th Catalan numbers as that is good enough up to $\text{poly}(n, 1/n)$ factors. We will also drop the factor 4 that arises in the transformation. Thus the number of beer connected proper interval graphs N on $n+3$ vertices is at least

$$\begin{aligned}
&\sum_{x=0}^{n} \sum_{y=0}^{(n-x)/2} \sum_{R_l, R_r \in S_{x,y}} g(R_l, R_r) \\
&\approx \sum_{x=0}^{n} \binom{n}{x} f(n,x) \sum_{y=0}^{(n-x)/2} \binom{n}{x} 4^y \\
&\approx \sum_{x=0}^{n} \binom{n}{x} f(n,x) 3^{n-x} \\
&\geq 6^n \sum_{x=0}^{n} \binom{n}{x} (1/6)^x \\
&= 6^n (7/6)^n = 7^n
\end{aligned}$$

Taking the log we see that $\lg(N) \geq n \lg 7 \approx 2.81n$.

On the other hand, for every beer proper interval graph, we may compute $g(R_l, R_r)$ which is exact if it is connected but is only an upper bound if not, thus we obtain the upper bound:

$$\begin{aligned}
&\sum_{x=0}^{n}\sum_{y=0}^{(n-x)/2}\sum_{R_l,R_r\in S_{x,y}} g(R_l,R_r)\\
&\approx \sum_{x=0}^{n}\binom{n}{x}f(n,x)\sum_{y=0}^{(n-x)/2}\binom{n}{x}4^y\\
&\approx \sum_{x=0}^{n}\binom{n}{x}f(n,x)3^{n-x}\\
&\leq 6^n\sum_{x=0}^{n}\binom{n}{x}(1/4)^x\\
&= 6^n(5/4)^n = (15/2)^n
\end{aligned}$$

Again taking the log we see that $\lg(N) \leq \lg(15/2)n \approx 2.91n$ $\square$

5.5.5 Improved Lower bound

In this section we will improve the lower bound attained in the previous section from $n\lg 7$ to $n\lg(6+\sqrt{2})$.

We begin with our counting identity:

$$C_{n+1} = \sum_{x=0}^{n}\sum_{y=0}^{(n-x)/2}\binom{n}{x}\binom{n-x}{2y}C_y$$

and rewrite it by switching the order of summation:

$$C_{n+1} = \sum_{y=0}^{n/2}\sum_{x=0}^{n-2y}\binom{n}{2y}\binom{n-2y}{x}C_y$$

With the interpretation of first choosing the $2y$ elements of $R_l \cup R_r \setminus R_I$, then choosing which set of R_l and R_r each of these element goes - which again must be a Dyck path on their own. Finally among the remaining elements we choose x of them to be in the intersection.

In this view, fix $R'_l = R_l \setminus R_I$ and $R'_r = R_r \setminus R_I$, with $R_I \subseteq [n] \setminus (R'_l \cup R'_r)$.

Define $T = \{(R'_l \cup R_I, R'_r \cup R_I); R_I \subseteq [n] \setminus (R'_l \cup R'_r)\}$ be the set of R_l, R_r that we can obtain.

Let $k_1, k_2 \ldots, k_{2y+1}$ be the composition defined by $R_I = [n] \setminus S$. Then for any smaller R_I, the effect on the partition is to split the parts k_i into smaller parts. Viewing part separately, we see that over all R_I, we obtain all partitions of each k_i independently.

Thus $\sum_{(R_l,R_r)\in T} g(R_l, R_r) = \Pi_{i=1}^{2y+1}(h(k_i))$ where $h(k_i)$ denotes the sum over all partitions of $(p_1, \ldots, p_j)$ of k_i elements where the value of each partition is of course $(p_1 + 1)(p_2 + 1) \ldots (p_j + 1)$.

$h(k)$ follows the recurrence $h(k) = \sum_{i=2}^{k+1} i \cdot h(k-i+1)$ by looking at the size of the last part of the composition. Furthermore it follows the recurrence $h(k) = 4h(k-1) - 2h(k+1)$ and has the close form formula $h(k) = \frac{(2+\sqrt{2})^{k+1} - (2-\sqrt{2})^{k+1}}{4\sqrt{2}}$. $h(k)$ is the sequence A003480 of OEIS [77].

As $\frac{(2+\sqrt{2})^{k+1} - (2-\sqrt{2})^{k+1}}{4\sqrt{2}} = \frac{(2+\sqrt{2})^{k+1}(1-\frac{1}{3\sqrt{2}})^{k+1}}{4\sqrt{2}} \geq \frac{(2+\sqrt{2})^{k+1}(1-\frac{1}{3\sqrt{2}}^2)}{4\sqrt{2}} = (2+\sqrt{2})^k(2-\sqrt{2})$ we obtain a nice form for

$$\sum_{(R_l,R_r)\in T} g(R_l, R_r) = \Pi_{i=1}^{2y+1}(h(k_i)) \geq (2+\sqrt{2})^n(2-\sqrt{2})^{2y+1}$$

.

Theorem 5.5.12. *To represent a beer proper interval graph G which is able to support* `adjacent` *and* `beer_distance` *will require at least $(\lg 6 + \sqrt{2})n - o(n) \approx 2.89n$ bits in the worst case. Furthermore, the lower bound cannot be greater than $n \lg 15/2 \approx 2.91n$ bits.*

Proof. We have already proven the upper bound in the previous section. For the lower bound, we again consider all Dyck path representation for proper interval graphs on $n+1$ vertices and transform them into connected proper interval graphs on $n+3$ vertices. The number of beer connected proper interval graphs is at least

$$\begin{aligned}
&\sum_{y=0}^{n/2}\sum_{x=0}^{n-2y}\binom{n}{2y}\binom{n-2y}{x}C_y g(R_l,R_r)\\
&\geq \sum_{y=0}^{n/2}\binom{n}{2y}2^{2y}(2+\sqrt{2})^n(2-\sqrt{2})^{2y}\\
&\geq (2+\sqrt{2})^n(1+2(2-\sqrt{2}))^n\\
&= (6+\sqrt{2})^n
\end{aligned}$$

□

5.5.6 Exact Lower Bound

Now we will derive a recurrence for the number of beer proper interval graphs, where solving the recurrence will give us the exact number for the lower bound. As we have shown, the lower bound αn satisfies $2.89 \approx \lg(6+\sqrt{2}) \leq \alpha \leq \lg 7.5 \approx 2.91$, so we have narrowed the range significantly already.

To derive the recurrence, we will decompose a Dyck path (which is in more or less a one-to-one correspondence to proper interval graphs) in such a way that it preserves the weights.

To see the one-to-one correspondence, we see that by the interval graph recognition algorithm of Booth and Leuker [15], the PQ-tree of a connected proper interval graph of the maximal cliques is a single Q node, so that there are at most two ordering of maximal cliques representing each graph (one order and the reverse of that order) (or we use the theorem of [53]). As each Dyck path gives a different sequence of maximal cliques, each graph can have at most two Dyck paths representing it.

For a particular proper interval graph, with twin vertex classes of sizes $k_1, \ldots, k_l$, the weight assigned to it is $\Pi_i(k_i+1)$. Now consider the following blocking scheme for the distance tree associated with the proper interval graph: start at the root and continue in level-order, add the vertices to the block until either: the vertex has a different parent, or the vertex is not a leaf. In this manner, we consider the root as a sibling of its left child.

Lemma 5.5.13. *In the above blocking scheme, two vertices u, v are in the same block if and only if they are twins.*

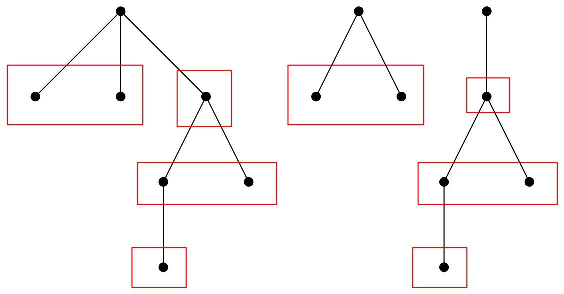


Figure 5.3: The twin vertex classes in the distance tree, and a way to decompose the tree

Proof. By Lemma 4.4.6, we see that the neighbourhood of any vertex v is an interval $[v_1, v_2]$ where $v_1 = \text{parent}(v)$ and v_2 is the rightmost child of the previous internal node of v in level order.

Thus two vertices $u < v$ are twins exactly when their neighbourhoods coincide, and this neighbourhood is $[v_1, v_2]$. They have the same v_1 exactly when they have the same parent. They have the same v_2 exactly when the previous internal node is the same, but that means v must be a leaf (as otherwise by definition, v is the previous internal node to v, and cannot be the previous internal node to u) and furthermore all the vertices between u,v are also leaves. Thus by definition, they would all be added to the same block as u. $\square$

We may look at this in the same way by replacing the root with a dummy root and dropping the original root as the first child of the dummy root. This blocking scheme is illustrated in Figure 5.3.

Note that we do not consider the dummy root as part of our blocks and we can also view this as deleting the dummy root and consider the roots of this new forest as siblings. The second is our proposed way to decompose the tree into two trees while preserving all the blocks. If we consider the balanced parenthesis view of the tree (without the dummy root), we see that the sequences are the same $()()|((())())$, but we cut it into two at the $|$. Precisely, $|$ is at the first spot in the sequence such that the excess is 0 and the next two parentheses are ((. In the language of Dyck paths, this is the first time the path touches the x-axis and the next two steps are both up-steps.

Let $C(n)$ be the n-th Catalan number and the number of Dyck paths of length $2n$. The above decomposes the path into two subpaths. Let $L(n)$ be the number of paths of length n for the subpath to the left of $|$. and $R(n)$ be the number of paths of length n of the subpath to the right. We will also abuse notation and use $L(n), R(n)$ as the set of Dyck paths of the respective forms. Thus we have the recurrence $C(n) = \sum_{k=0}^{n} L(k)R(n-k)$. As a sanity check, we show that this indeed is a recurrence for $C(n)$.

What is $L(n)$? Let $2l$ be the first time that the path touches the x-axis. As we cannot have up-up steps immediately following, the steps to the right of this point must be a sequence of up-down steps. The path before we touch the x-axis for the first time is a irreducible dyck path, which is $C(l-1)$. Thus $L(n) = \sum_{l=0}^{n} C(l-1)$.

$R(n)$ are dyck paths that begin with two up-steps. Since all dyck paths either begin with up-up or up-down, we have $R(n) = C(n) - C(n-1)$, where $C(n)$ counts those paths that begin with up-up and $C(n-1)$ counts those paths that begin with up-down.

It is easily seen that by expanding out these definitions, we do indeed have the recurrence $C(n) = \sum_{k=0}^{n} L(k)R(n-k)$ - if we do expand out the definitions, then we obtain a telescoping sum of $C(n) - C(n-1) + C(n-1) - C(n-2) \cdots + C(0) - C(-1)$ on the right hand side.

Now we consider the weighted versions. Let $\bar{C}(n)$ be the sum of all Dyck paths with our weighting system. Similarly for $\bar{L}(n)$ and $\bar{R}(n)$. Because we preserve all the blocks with our split, we have the same recurrence $\bar{C}(n) = \sum_{k=0}^{n} \bar{L}(k)\bar{R}(n-k)$, which holds for $n \geq 1$. For $n = 0$, we see that $\bar{L}(0) = 0, \bar{R}(0) = 1$ and $\bar{C}(0) = 1$. Now it remains to compute $\bar{L}(n)$ and $\bar{R}(n)$.

Lemma 5.5.14. *$\bar{L}(n) = \sum_{l=0}^{n} \bar{C}(n-l-1)(l+2)$ and $\bar{R}(n) = \bar{C}(n) - \sum_{l=1}^{n} (l+1)\bar{R}(n-l)$.*

Proof. We split a Dyck path in $L(n)$ as above: an irreducible Dyck path followed by a sequence of up-downs. Suppose that the irreducible Dyck path has length $2(n-l)$, then the top level has a block of size $l+1$, as there are l up-downs and the 1 node contributed by the irreducible Dyck path. The remainder of the Dyck path is of length $2(n-l-1)$ and thus contributes $\bar{C}(n-l-1)$.

This is illustrated in Figure 5.4. As the first part of the Dyck path is irreducible, it is a rooted tree, and the block at level 1 contains $l+1$ nodes. The remainder of the blocks are exactly the same as those in the first node's subtree.

We again decompose $R(n)$ as all path minus those that begin with up-down. Let l be the number of up-downs that begins the path. These corresponds to leaves that begin the tree and are in a block together. The rest of the path must begin with up-up and is thus a path in $R(n-l)$. Therefore, we have $\bar{R}(n) = \bar{C}(n) - \sum_{l=1}^{n} (l+1)\bar{R}(n-l)$.

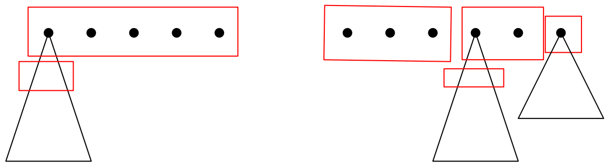


Figure 5.4: Decomposition of Dyck paths (as viewed as trees) of the forms L and R

This is illustrated in Figure 5.4 in the second forest. There are 3 leaves that begin the tree, corresponding to 3 up-downs that starts the path, and creates a block of size 3 (with weight 4). The rest of the forest must begin with an up-up and belong to $R(n-3)$. $\square$

Now consider the following generating functions. Let $f(x) = \sum_{n\geq 0} \bar{C}(n)x^n$, $g(x) = \sum_{n\geq 0} \bar{L}(n)x^n$ and $h(x) = \sum_{n\geq 0} \bar{R}(n)x^n$. The above recurrences says that these generating functions are linked and that we have a very nice closed form for f.

Lemma 5.5.15. *Let $b(x) = (1-x)^{-2}$. Then we have $f = gh + 1$, $g = f \cdot (b-1)$ and $h = f/b$. Finally $f = \left(1 - \sqrt{1-8x+4x^2}\right) / (4x - 2x^2)$.*

Proof. We first note that $b(x) = \sum_{n\geq 0}(n+1)x^n$. This is easily seen as $b(x)$ is the derivative of $(1-x)^{-1} = \sum_{n\geq 0} x^n$.

Next we note that as $\bar{C}(n) = \sum_{k=0}^{n} \bar{L}(k)\bar{R}(n-k)$ for $n \geq 1$, we obtain $f = gh + 1$. The constant term accounts for the initial conditions.

Next we expand the recurrence for $\bar{L}$ to obtain:

$$\begin{aligned} g &= \sum_{n=0}^{\infty} \bar{L}(n)x^n = \sum_{n=0}^{\infty}\sum_{l=0}^{n} \bar{C}(n-l-1)(l+2)x^n = \sum_{l=0}^{\infty}\sum_{n=l}^{\infty} \bar{C}(n-l-1)(l+2)x^n \\ &= \sum_{l=0}^{\infty}(l+2)x^{l+1}\sum_{n=l}^{\infty} \bar{C}(n-l-1)x^{n-l-1} = \sum_{l=1}^{\infty}(l+1)x^l f \\ &= f \cdot (b-1) \end{aligned}$$

Expanding the recurrence for $\bar{R}$ we obtain:

$$\begin{aligned}
h &= \sum_{n=0}^{\infty} \bar{R}(n)x^n = \sum_{n=0}^{\infty} \bar{R}(n)x^n - \sum_{n=0}^{\infty}\sum_{l=1}^{n} \bar{R}(n-l)(l+1)x^n \\
&= f - \sum_{l=1}^{\infty}\sum_{n=l}^{\infty} \bar{R}(n-l)(l+1)x^n = f - \sum_{l=1}^{\infty}(l+1)x^l \sum_{n=l}^{\infty} \bar{R}(n-l)x^{n-l} \\
&= f - (b-1)h
\end{aligned}$$

Collect terms and we obtain $f = bh$.

Lastly, we have

$$\begin{aligned}
0 = gh + 1 - f &= f^2 \frac{(1-x)^{-2} - 1}{(1-x)^{-2}} + 1 - f \\
&= f^2(1-(1-x)^2) - f + 1 = (2x - x^2)f^2 - f + 1
\end{aligned}$$

Apply the quadratic formula and taking the negative root, we obtain the desired $f = \left(1 - \sqrt{1 - 8x + 4x^2}\right) / (4x - 2x^2)$ □

We note that the sequence A108524 of OEIS [77] has the same generating function and thus $\bar{C}(n)$ is exactly A108524. Furthermore, we can calculate the asymptotics of $\bar{C}(n) = (4 + 2\sqrt{3})^n \cdot \text{poly}(n, 1/n)$, using methods such as Thm 2.11 of [57]. Thus we can finally prove our desired lower bound for the number of beer proper interval graphs.

Theorem 5.5.16. *The number of beer proper interval graphs on n vertices is asymptotically $(4+2\sqrt{3})^n \cdot poly(n, 1/n)$. Therefore to represent a beer proper interval graph G which is able to support* `adjacent` *and* `beer_distance` *will require at least* $\lg(4 + 2\sqrt{3})n - o(n) \approx 2.9n$ *bits in the worst case.*

Proof. Consider a connected proper interval graph. The weight assigned to it is the weight of the distance tree after we add the dummy root and drop the real root as the first child of the dummy root.

Ignoring the dummy root, this is a particular forest on n nodes and thus is counted in $\bar{C}(n)$. Hence $\bar{C}(n)$ is an upper bound on the number of beer connected proper interval graphs.

Conversely, if we simply delete the root of the distance tree, we obtain a Dyck path on $n - 1$ vertices, and thus the weight is counted in $\bar{C}(n-1)$. The root of the tree can only increase the weight (either by being in a block by itself or increasing the size of the first block by 1). Thus $\bar{C}(n-1)$ is a lower bound.

Since $\bar{C}(n) = (4+2\sqrt{3})^n \cdot \text{poly}(n, 1/n)$, it does not change asymptotically if we change the number of vertices by 1 (as that only induces a $\text{poly}(n, 1/n)$ factor). Hence the number of beer connected proper interval graphs on n vertices is asymptotically $(4 + 2\sqrt{3})^n \cdot \text{poly}(n, 1/n)$.

To convert this to count all proper interval graphs, we note that for any proper interval graph, the blocking scheme is an upper bound on the number of beer vertex patterns (as we are not considering a number of automorphisms in our count - the ones that swap isomorphic components), and thus $(4+2\sqrt{3})^n \cdot \text{poly}(n, 1/n)$ is an upper bound. Clearly, connected proper interval graphs is a subset of all proper interval graphs and $(4+2\sqrt{3})^n \cdot \text{poly}(n, 1/n)$ is a lower bound. □

5.6 Discussion

In this chapter, we studied the shortest beer path problem in interval graphs and proper interval graphs. For upper bounds, we constructed a $3n + o(n)$ bit data structure for beer proper interval graphs, which can answer queries in $O(f(n) \lg n)$ time for $f(n) = \omega(1)$ (ex. $\lg\lg n$). For beer interval graphs, we constructed a $n \lg n + o(n \lg n) + O(|B| \lg n)$ bit data structure answer queries in $O(\lg^\epsilon n)$ time. To do this, we showed that the optimal beer vertex can be found in a rectangular region of the distance tree, and we search for it using orthogonal range search data structure.

For lower bounds, we enumerated the number of beer graphs. We first defined a natural notion of isomorphism between beer graphs, and used results from group theory to explicitly compute the number of distinct beer vertex patterns a graph G can have from the automorphism group of G. Using this, we were able to derive a formula for beer proper interval graphs. We used two approaches to finally conclude that the lower bound for beer interval graphs is $n \lg n$ bits and for beer proper interval graphs is $\lg(4 + 2\sqrt{3})n$ bits.

Our results are not optimal, and thus leave quite a bit of room for future research. On the space side, our data structure for beer proper interval graphs has a gap between $3n$ and $\lg(4 + 2\sqrt{3})n$ bits. For beer interval graphs, gap is the term $O(|B| \lg n)$ required for the range search data structures. This dependence on $|B|$ could potentially be removed, perhaps using similar techniques as Section 5.3. On the query time side, both beer proper interval graphs and beer interval graphs uses orthogonal range search, which has a running time $O(\lg^{\epsilon} n)$. If a different method can be found, it may be possible to reduce the time complexity as well.

Lastly, there are other classes of graphs to be considered. As far as we know, the only class of graphs considered are outerplanar graphs [7], and what is considered here. Other classes that could potentially be considered include path graphs, chordal graphs, planar graphs among others.

Chapter 6

Dynamic Interval Graphs

In this chapter, we will study the data structure problem on dynamic interval graphs. In this setting, we are allowed to make changes to the graph, specifically, we will allow the addition and deletion of vertices as defined by an interval. We will not explicitly allow the insertion and deletion of edges as the edges are defined implicitly through the intersection structure.

The chapter is organized as follows: we begin with a review of relevant previous works in Section 6.1 and an overview of data structures that we will be using in Section 6.2. We will first consider the navigation queries `adjacent`, `degree`, `neighborhood` in Section 6.3. Next we will consider various different models of dynamic operations and the ability to support the `distance` operation: first is insertions and deletions in proper interval graphs in Section 6.4, then we consider either only insertions or only deletions in interval graphs in Section 6.5, then we consider the case of allow both insertions and deletion, but with the restriction that we are given all the operations in advance in Section 6.6 and finally, we relax all the restrictions and allow insertions and deletions in Section 6.7. Lastly, we will summarize this chapter and give avenues for future work in Section 6.8.

6.1 Previous Work

One reason why the result of Chapter 4 is interesting is that, to achieve linear space, it is not possible to store the edges explicitly; unlike planar graphs, the number of edges in an interval graph can possibly be quadratic. Instead, researchers focus on designing data structures over the intervals represented by the graph. Thus, this provides answers to the question about whether one can get more efficient solutions to graph problems when graphs are provided implicitly. In other words, this is an instance in which graphs cannot be written down explicitly, and you want data structures that use $\tilde{O}(n)$ space or algorithms that work in $\tilde{O}(n)$ time, where n is the number of vertices, instead of explicitly constructing all edges. Other instances include the work of Alman et al. [4], who initiated the investigation on what kind of geometric graphs possibly allow faster, $O(n^{1+o(1)})$-time algorithms for various problems in spectral graph theory, instead of first spending quadratic time on explicitly writing down the graph and then applying linear-time algorithms. The work of Munro and Sinnamon [64] on representing distributive lattices also avoided explicitly storing the up to $n \lg n$ edges of the transitive reduction of a distributive lattice on n elements, and instead extended the ideal tree structure of a distributive lattice to design an $O(n)$-word representation that supports finding meets and joins in $O(\lg n)$ time.

In this chapter, we hope to provide some answers to the following question: "What graphs allow more efficient solutions to dynamic graph problems when the graphs are provided implicitly?"

The field of dynamic graph algorithms has seen extensive study. The goal is to answer queries such as connectivity (are two vertices in the same connected component), shortest paths, matching, among others on an arbitrary graph undergoing edge/vertex insertions and deletions (though it is often assumed that either no vertex insertions/deletions occur or they only happen to degree 0 vertices). For connectivity, the first algorithm is by Henzinger and King [47] and the best known algorithm is by Huang et al. [48]. We refer the reader to the talk of Hanauer et al. [41] and the survey paper of Henzinger [46].

6.2 Preliminaries

We will restate our definitions of interval graphs.

Definition 2.1.4. A graph G is an ***intersection graph*** if we may associate every vertex v with a set s_v such that for any two vertices, $(u, v) \in E$ if an only if $s_u \cap s_v \neq \emptyset$. We say that the family of sets is an ***intersection model*** for the graph.

An ***interval graph*** is intersection graph of a set of intervals on the real line. Therefore for an interval graph G, we may find a set of intervals $\mathcal{I}(G)$ as the intersection model (or more descriptively, an *interval representation* of G). As we will be inserting and deleting vertices as represented by an interval, we will fix a particular intersection model for G.

A ***proper interval graph*** is an interval graph where we may associate each vertex with an interval so that no interval is completely covered by another.

We define the following operations over an interval graph G:

- `insert`(v): add to G a vertex v given by the interval I_v.
- `delete`(v): deletes from G a vertex v given the interval I_v.
- `distance`(u, v): returns the distance between two vertices in G.

We note that rather than inserting vertices and edges independently as in Definition 2.1.3, we instead only insert vertices as presented by a set, and implicitly insert all the edges that would exist from the intersection structure.

Now we will restate some of the results proved in Chapter 4 for the static case.

For each interval v, we define the parent relationship `parent`(v) as the vertex u such that

$$u = \arg\min \{l_w \mid r_w \geq l_v\} \tag{6.1}$$

This is simply the expanded form of the definition given in Section 4.4.

Our goal is to represent the graph using $O(n \lg n)$ bits (i.e. compactly). As we cannot order the components as in 4.4 (because we do not control the order of the components and because the equivalence class structure is not dynamic), we will instead store one tree per component as defined in subsection 4.4.

Thus the distance tree we will use is rather a forest, as defined below:

Definition 6.2.1. Let G be an interval graph, with a fixed interval representation. The ***distance tree*** $T(G)$ is defined under the parent relationship `parent`(v). For every vertex v, we order the children of v in order of the left end point of the vertices. That is, if u, w are two children of v with $l_u < l_w$ then u is to the left of w.

If the graph is disconnected, then we will have a forest instead. We have one tree per vertex v where `parent`$(v) = v$. Furthermore when we refer to the distance tree $T(G)$ of G in the context of a vertex v if G is disconnected, it is understood that we refer to the tree in the forest that contains v.

For an ordinal (or cardinal) tree T, and a node v, we denote the quantity $\texttt{node\_rank}_X^T(v)$ as the index of v in the X traversal of T, where X could be $\texttt{LEVEL}, \texttt{PRE}, \texttt{POST}$ indicating a breadth-first traversal, pre-order traversal and post-order traversal respectively. [1]

Recall that by Lemma 4.4.1, if $u < v$ (i.e. $l_u < l_v$) then $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$ and vice versa. We note that by the properties of all of the above traversals, if $\texttt{depth}(u) = \texttt{depth}(v)$, then the relative ranks of u, v are the same in all traversals. That is $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v) \Leftrightarrow \texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$, and the same with $\texttt{node\_rank}_{\texttt{PRE}}$.

Now we recall the way we calculated a shortest path and the distance in an interval graph, using Lemma 4.4.2, which we will restate here:

Lemma 4.4.2. *Let G be an interval graph with distance tree T. Let u, v be two vertices of G with $\texttt{node\_rank}_{\texttt{LEVEL}}(u) > \texttt{node\_rank}_{\texttt{LEVEL}}(v)$. Consider the node to root path of u as $u = u_1, \ldots, u_k = r$. Let i be the first index where $l_{u_i} \le r_v$. Then a shortest path from u to v is $u = u_1, \ldots, u_i, v$.*

Furthermore, $\texttt{depth}(u_i)$ is either $\texttt{depth}(v) - 1$ or $\texttt{depth}(v)$ or $\texttt{depth}(v) + 1$.

We note that in the proper interval graph case, since the entire graph is encoded in the distance tree, we may state it more succinctly as:

Lemma 6.2.2. *Let G be a proper interval graph with distance tree $T(G)$ and u, v be two vertices of G with $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$. Then[2] $\texttt{distance}(u, v) = \texttt{depth}(v) - \texttt{depth}(u) + \mathbf{1}\left(\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)\right)$, where the last term evaluates to 1 if the expression inside the brackets is true and 0 otherwise.*

Proof. Let $k_1 = \texttt{depth}(u) \le \texttt{depth}(v) = k_2$. Let the path to the root from v be named $v = v_{k_2}, \ldots, v_0 = r$. Let i be the first (i.e. largest) index where $l_{v_i} \le r_u$ so that they are adjacent. We know that a shortest path would be $v, \ldots v_i, u$. Furthermore, we know that $i = \texttt{depth}(u)$ or $i = \texttt{depth}(u) + 1$. In the first case, $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$, as otherwise u would be between v_i and v_{i+1} and would be adjacent to v_{i+1}. In this case the distance is $k_2 - k_1 + 1$ which the formula evaluates correctly.

In the second case, if $i = \texttt{depth}(u) + 1$, then $\texttt{node\_rank}_{\texttt{LEVEL}}(u) \ge \texttt{node\_rank}_{\texttt{LEVEL}}(v)$, as otherwise, u would not be adjacent to v_i. In this case, the distance is $k_2 - k_1$ and again the formula evaluations correctly. $\square$

[1] We will omit the superscript when the tree being referred to is clear.

[2] If the two vertices belong to the same tree; otherwise they are in different components and the distance is ∞

Some other dynamic data structures we are using are some more specialized trees. First is a balanced binary search tree [75] that allows batch deletions by delaying rebalancing until the next insert (a naive approach in AVL [3] trees would require $O(\lg n)$ on each deletion due to rebalancing).

Lemma 6.2.3. *There exists a balanced binary search tree using $O(n)$ words of space of height $O(\lg n)$, which supports insertions in $O(\lg n)$ plus $O(1)$ amortized time, and deletion in $O(1)$ time plus the time ($O(\lg n)$) needed to find the node.*

Next is a dynamic weighted tree structure which allows the computation of weighted path lengths in the tree.

Lemma 6.2.4. *(Top Tree [5]) Let T be a forest. A top tree data structure on T occupies $O(n)$ words of space and supports the following operations* [3] *in $O(\lg n)$ time:*

- `link`(u, v), *where u and v are in different trees, links these trees by adding the edge (u, v) to our dynamic forest.*
- `cut`(e), *removes the edge e from the forest.*
- `update_weight`(e, w), *update the weight of the edge e to w and return the previous weight.*
- `weighted_distance`(u, v), *returns the weight of the path between u and v.*
- `anc`(u, v, d), *returns the first node in the tree on the path from u to v at distance at least d from u.*

We will also need the dynamic form of the orthogonal range search data structure. The following is a result by Mortensen [58]

Lemma 6.2.5. *There is a data structure occupying $O(n \lg n / \lg\lg n)$ words of space which supports the insertions and deletion of 2-dimensional points in $O(\lg n)$ time and the 2-dimensional orthogonal range reporting query in $O(\lg n + k)$ time where n is the current number of points in the data structure and k is the number of points reported.*

[3] A top tree is flexible and supports many more operations than listed here, but these are the only ones we will need

6.3 Navigational Queries

We will now handle the relatively easy case when the only operations we need to consider are `insert`, `delete`, `adjacent`, `neighborhood`, `degree`, `spath`. We will adapt the data structure of Acan et al. [2] to to easily handling insertions and deletions.

First we note that their data structure reduces the endpoints of the interval to $[1, 2n]$. This is allowed in the static case since we can choose the interval representation, but not in the dynamic case since the intervals are given to us.

In this model, whenever the input to a query is a vertex, it will actually be the interval corresponding to that vertex.

Therefore, we will store the intervals $[l_v, r_v]$ in two balanced binary search trees. First with the left end points as keys, and the second with right end points as keys. At each node, we will also store the subtree sizes. Furthermore, for the tree on the left endpoints, we will store the maximum right endpoint in the subtree and a pointer to the node containing this endpoint, and symmetrically, on the tree on the right endpoints, we store the minimal left endpoint in the subtree, and the pointer to the node.

We may then easily translate `insert` and `delete` to inserting and deleting in trees.

Now `adjacent` is trivial, since we are given the intervals of the two vertices.

For `degree`, we are given an interval $[l_v, r_v]$. As in Acan et al., we will find the number of vertices not adjacent to v. This requires to find the number of intervals I_u with $r_u < l_v$ and the number of intervals I_w with $l_w > r_v$. The first can be obtained by searching for the value l_v in the tree of right endpoints. The nodes with $r_u < l_v$ are potentially on the path and in complete subtrees as left children of nodes on the path. Similarly for the nodes with $l_w > r_v$.

As we explained in subsection 4.3 in the abstraction Theorem 4.3.7, `neighborhood` are either vertices u with $l_v \leq l_u \leq r_v$, which is a sequence of nodes in the in-order traversal of the tree on the left endpoints, or vertices w with $l_w < l_v$ and $r_w > l_v$. This second case we find the nodes by traversing the tree using the maximum right endpoints in each subtree.

Finally, by 4.4.2, to compute a shortest path, it suffices to be able to support `parent`. Given a vertex v, we need to find the vertex u satisfying equation 6.1. To find u, we search for the value l_v in the tree on the right endpoints. Among the nodes with right endpoint larger than l_v, we find the node with minimal left endpoint. This takes $O(\lg n)$ time.

Thus we have:

Theorem 6.3.1. *Let G be an interval graph with a fixed interval representation $\mathcal{I}$. Then we may store G using $O(n \lg n)$ bits supporting* `insert`, `delete` *in $O(\lg n)$ time,* `adjacent` *in $O(1)$ time,* `degree` *in $O(\lg n)$ time and* `neighborhood`, `spath` *in $O(\lg n)$ time per neighbour or vertex on the path.*

6.4 Dynamic Proper Interval Graph

In this section, we will be studying proper interval graphs. In particular, we would like to support `insert`, `delete`, `spath` and `distance`. To do this, we will maintain the distance tree under insertions and deletions of vertices.

As we have restricted the problem we will also naturally modify the `insert`(v) operation. If the interval $[l, r]$ inserted is incompatible with the proper interval graph - that is it either covers or is covered by another interval, we discover this and abort the operation.

6.4.1 Distance Tree Modifications

The main issue is that the distance tree $T(G)$ in general has unbounded degree, which makes updates difficult since we may need to updated the parent of an unbounded number of nodes in the case that the parent were to be deleted. To alleviate this, we will apply the well known isomorphism between ordinal trees and binary trees to ensure that the tree is binary. To preserve path lengths, sibling edges in the binary tree will have weight 0, while parent edges in the binary tree will have weight 1.

Definition 6.4.1. Formally the isomorphism that we will use is the following transformation. Let T be an ordinal tree on n nodes. Define the weighted binary tree $\mathbf{T_B}$ also on n nodes as follows: For each vertex v in T_B, the left child of v is its left sibling in T, the right child of v is its right most child in T. Left child edges have weight 0, right child edges have weight 1.

Using this convention, whenever we add a vertex as the left child of another vertex, it is implicit that we also set the weight of that edge to 0, similarly for right child edges. This transformation preserves post order traversal. Furthermore, when we talk about node depth or path length in T_B, we refer to weighted node depth or weighted path length, respectively. Hence, any path length in $T(G)$ is also invariant under this transformation.

Thus Lemma 6.2.2 still holds under this transformation. Furthermore, we will use `node_rank`$_{\texttt{LEVEL}}(u)$ to refer to the level-order position of u in $T(G)$ before the transformation as the level-order position no longer has any meaning after the transformation. As concepts are more easily stated on $T(G)$, we will mainly use it for stating relationships between vertices, but straightforwardly translate the operations on $T_B(G)$ which we will maintain.

Under this transformation, we also immediately have the analogous notion of ancestors. For a vertex v, the node $u = \texttt{anc}_T(v, d)$ is the ancestor of v at depth d in T. The node u in T_B is the closest ancestor of v at depth d, which in a top tree would be $\texttt{anc}_T(v, r, d)$ (as the edges may have 0 weight there are multiple ancestors at each depth).

6.4.2 Fully Dynamic Proper Interval Graphs

We are now ready to describe our data structure. For a proper interval graph G, with intervals $\mathcal{I}$, we will maintain the following:

- A top tree of $T_B(G)$. For each component, we store a variable indicating the root r of that component.
- A mapping between the vertices v of G and the interval $I_v = [l_v, r_v]$.
- A mapping from the end points of intervals to the vertex itself. Note that no two intervals can share left end points nor right end points in a proper interval graph, but the left end point of one interval can be the right end point of another.
- All the left end points of all the intervals.
- All the right end points of all the intervals.

For the last 4 items, we will use a balanced binary tree such as an AVL tree [3], so that searches can be done in $O(\lg n)$ time. For the last 2 items, this also allows us to support successor and predecessor queries, denoted by $\texttt{pred}_L/\texttt{succ}_L$, on the left end points and $\texttt{pred}_R/\texttt{succ}_R$ on the right end points. The total space is $O(n)$ words.

Thus for any vertex v, $\texttt{depth}(v)$ is obtained by $\texttt{weighted\_distance}_{T(G)}(r, v)$ in $O(\lg n)$ time. Furthermore $\texttt{anc}(v, d)$ may also be obtained in $O(\lg n)$ time using the ancestor operation on top trees. Now we show how to immediately translate the distance calculation from $T(G)$ into our data structure.

Lemma 6.4.2. *The data structures in this section can compute* $\texttt{distance}(u, v)$ *in* $O(\lg n)$ *time given two vertices,* u, v *of* G.

Proof. In the case that $\texttt{depth}(u) = \texttt{depth}(v)$, we retrieve the end points. Assume without loss of generality that $l_u > l_v$. Since $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v) \Leftrightarrow \texttt{node\_rank}_{\texttt{LEVEL}}(u) > \texttt{node\_rank}_{\texttt{LEVEL}}(v) \Leftrightarrow l_u > l_v$, by Lemma 6.2.2 we have that $\texttt{distance}(u, v) = 1$.

Now suppose that $\texttt{depth}(u) > \texttt{depth}(v)$. By the property of a breadth-first traversal, we also have $\texttt{node\_rank}_{\texttt{LEVEL}}(u) > \texttt{node\_rank}_{\texttt{LEVEL}}(v)$. Using the top tree, we find $w = \texttt{anc}(u, \texttt{depth}(v))$ in $O(\lg n)$ time. Then $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v) \Leftrightarrow \texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v) \Leftrightarrow l_w > l_v$, so a comparison between l_w and l_v is sufficient to apply Lemma 6.2.2. $\square$

Now we consider maintaining the distance tree $T(G)$ (conceptually) and $T_B(G)$ (concretely) under updates. To do so, we first characterize the parent relationship in $T_B(G)$.

Lemma 6.4.3. *Let G be a proper interval graph with distance tree $T_B(G)$. Let v be a vertex. If v is not a root of one of the components, then $\texttt{parent}_{T_B(G)}(v)$ is*[4]

$$\arg\min\{l_w \mid l_w \geq l_v\} \cup \{r_w \mid r_w \geq l_v\} \tag{6.2}$$

If two vertices u, w have end points such that $r_u = l_w$, break ties in the above quantity by treating $l_w < r_u$.

Furthermore, let v' be the vertex where $l_{v'} = \texttt{pred}_L(l_v)$ the predecessor of v. Then v is the root of its component if and only if v' is not adjacent to v.

Proof. First suppose that v' is adjacent to v. Then, since $l_{v'} < l_v$, we have $r_{v'} \geq l_v$, and hence, v' is a candidate in the parent (in $T(G)$) relationship of v. Therefore, v would have a parent in $T(G)$ and thus would not be the root.

Conversely, let p denote the parent of v in $T(G)$. If $v' \neq p$, we have $l_p < l_{v'} < l_v \leq r_p < r_{v'}$. The first and third inequality comes from p being the parent of v, while the others comes from the fact that G is a proper interval graph. Thus v' is adjacent to v.

Now suppose that v is not the root of its component. By the construction of $T_B(G)$, v's parent in $T_B(G)$ is either its right sibling or, if it has no right sibling, its parent in $T(G)$. Let $u = \arg\min\{l_w \mid l_w \geq l_v\} \cup \{r_w \mid r_w \geq l_v\}$. Suppose that u is obtained from the first set which consists of left end points. Then as there are no right end points between l_v and l_u, they have the same parent in $T(G)$. Since $l_u = \texttt{succ}_L(l_v)$, u must be the immediate

[4] arg min of a set of values computed from a set of objects (i.e. l_w computed from the vertices w) returns the object achieving the minimum value, rather than the value itself for min.

right sibling of v. If u is obtained from the second set which consists of right end points, then as G is a proper interval graph, $u = \arg\min\{l_w \mid r_w \geq l_v\}$, so u is the parent of v in $T(G)$. □

We will first consider insertions. Let w be the vertex with interval $I_w = [l, r]$ be our insertion candidate. We will accomplish this in two steps: first, check that w contains or is contained by some interval in G. If so, we may stop immediately. Then, determine the links of $T_B(G)$ that need to be updated.

For step 1, we only need to check the containment between the two immediate predecessor and successors of w. This can be done in $O(\lg n)$ following from this lemma:

Lemma 6.4.4. *Let G be a proper interval graph with intervals $\mathcal{I}$. Let $w = [l, r]$ such that $l \neq l_v$ is the not the left end point of any interval $I_v \in \mathcal{I}$. Let v be the vertex such that $l_v = \texttt{pred}_L(l)$ and u be the vertex such that $l_u = \texttt{succ}_L(l)$.*

Then w is contained in some interval $I_{v'}$ if and only if w is contained in I_v, and w contains some interval $I_{u'}$ if and only if w contains I_u.

Proof. By assumption, $l_v \neq l_w \neq l_u$. As v is the predecessor of w, $l_{v'} \leq l_v$. If w is contained in $I_{v'}$, then $l_{v'} \leq l_v < l_w < r_w \leq r_{v'} < r_v$, so w is also contained in I_v. Conversely, we choose $v' = v$.

Now suppose that w contains some interval $I_{u'}$. Then we have $l_w < l_u \leq l_{u'} \leq r_u < r_{u'} < r_w$, so w contains u. □

We will now assume that $G \cup \{w\}$ is a proper interval graph. It remains to update parent links in $T_B(G)$.

Lemma 6.4.5. *$O(1)$ links need to be updated to transform $T_B(G)$ to $T_B(G \cup \{w\})$.*

Proof. By Lemma 6.4.3 for a vertex v, the parent in $T_B(G)$ is given by equation 6.2. By adding l_w and r_w, the only links we need to add is $\texttt{parent}_{T_B(G\cup\{w\})}(w)$, and whenever w is the result of equation 6.2. Furthermore, as roots do not have parents, if w becomes the new root of some component, the old root would need to relink as well.

Thus by the analysis above, at most 4 links need to be updated. We note that to compute the new parent of any node, equation 6.2 can be calculated using $\texttt{succ}_L(l_v)$ and $\texttt{succ}_R(l_v)$, then taking the minimum of the result.

To be complete, we will explicitly state the vertices that need to relink. If w is the new root of some component, then the old root is $l_r = \texttt{succ}_L(l_w)$ if r is adjacent to w.

Otherwise, w is in a component by itself. In the case that r is adjacent to w, r will need to recalculate its parent link.

If w is not the new root of some component, then we calculate the parent of w. The two children of w are the vertices whose left end point immediately proceeds l_w and r_w.

Let u, v be the vertices such that $l_u = \texttt{pred}_L(l_w)$ and $l_v = \texttt{pred}_L(r_w)$. We may need to relink u and v as they may now be children of w. □

Lemma 6.4.6. *The transformation from $T_B(G)$ to $T_B(G \cup \{w\})$ take $O(\lg n)$ time and thus* `insert` *has time complexity $O(\lg n)$.*

Proof. `insert` first checks that w is consistent with the rest of the intervals. This takes $O(\lg n)$ time. The transformation of the distance trees requires the relinking of $O(1)$ links, which takes $O(\lg n)$ time. Adding w to the maps between vertices and end points and adding the end points of w to the trees require $O(\lg n)$ time.

Thus in total, `insert` requires $O(\lg n)$ time. □

The `delete` operation is in essence the reverse of `insert`. Furthermore, we do not need to check that the new vertex is compatible with the rest of the intervals.

Lemma 6.4.7. *Removing a vertex w from $T_B(G)$ requires the relinking of at most $O(1)$ links.*

Proof. The proof will essentially be the same as in the insertion case.

Clearly the children of w will need to be relinked. The parent pointer of w will need to be removed if it exists as well. Lastly, if w would disconnect the tree, then we need to find the new root of the new component as well. This is $O(1)$ links.

Again to be complete, we will find the vertices that will require relinking.

First suppose that w would disconnect the tree. Then the new root is u such that $l_u = \texttt{succ}_L(l_w)$. By Lemma 6.4.3, we can detect that this is a new root by checking the adjacency between u and its predecessor v found by $l_v = \texttt{pred}_L(l_w)$.

The children of w can be found by $\texttt{pred}_L(l_w)$ and $\texttt{pred}_L(r_w)$. □

Lemma 6.4.8. *Let G be a proper interval graph with distance tree $T_B(G)$. Then* `delete`(w) *can be done in $O(\lg n)$ time.*

Proof. The transformation of the distance trees requires the relinking of $O(1)$ links, which takes $O(\lg n)$ time. Removing w from the maps between vertices and end points and removing the end points of w from the trees require $O(\lg n)$ time.

Thus in total, `delete` requires $O(\lg n)$ time. □

Having shown how to support `insert`, `delete` and `distance`, we have the following theorem.

Theorem 6.4.9. *A proper interval graph G can be represented in $O(n)$ words of space, where n is the number of vertices currently in G, to support* `insert`, `delete`, `distance` *in $O(\lg n)$ time, and* `spath` *in $O(\lg n)$ time per vertex on the path.*

Proof. The previous lemmas shows how to support `insert`, `delete` and `distance`. By Lemma 4.4.2, we may obtain a shortest path by computing $\texttt{parent}_{T(G)}$. By definition of $T(G)$ and equation 6.1, for a vertex v, $\texttt{parent}_{T(G)}$ is found by $\texttt{succ}_R(l_v)$, and thus `spath` can be supported as well in $O(\lg n)$ time per vertex on the path. □

6.4.3 Computation of Distance on Two Additional Intervals

In this section, we will consider the an additional query over our structures in $O(\lg n)$ time, which is used in Section 6.5: Given two vertices u, v (represented by intervals) not in the proper interval graph G, compute $\texttt{distance}(u, v)$ in $G \cup \{u, v\}$ without requiring $G \cup \{u, v\}$ to be a proper interval graph.

Let G be a proper interval graph with intervals $\mathcal{I}$. Let $u = [l_u, r_u], v = [l_v, r_v]$ be two intervals. $G \cup \{u, v\}$ is not necessarily a proper interval graph, but is an interval graph. We would like to find the distance between these two vertices in the interval graph.

Definition 6.4.10. For a proper interval graph G and two vertices u, v not in G defined by an interval representation, denote

$$\texttt{distance}_G(u, v) = \texttt{distance}_{G \cup \{u,v\}}(u, v)$$

Lemma 6.4.11. *Let G be a proper interval graph, and $u = [l_u, r_u], v = [l_v, r_v]$ be two vertices not in G. Suppose that $r_u < l_v$, and define $u' = [r_u, r_u], v' = [l_v, l_v]$, then*

$$\texttt{distance}_G(u, v) = \texttt{distance}(u', v').$$

We note that by symmetry we assume $l_u < l_v$. If $r_u > l_v$ the interval intersect and the distance is 1. The above lemma ignores this trivial case. The above lemma also says that we may ignore the intervals and use just the points representing the left/right endpoints of the intervals instead. Thus the distance does not change if we were to shift the left end point of u or the right end point of v.

Proof. We will show that $\texttt{distance}_G(u, v) = \texttt{distance}_G(u, v') = \texttt{distance}(u', v')$.

Let $T_{x,y}$ be the distance tree of the interval graph $G \cup \{x, y\}$ where $x, y \in \{u, v, u', v'\}$.

We first consider the differences between $T_{u,v}, T_{u,v'}, T_{u',v'}$. For the first two, the tree on the vertices w with $l_w < l_v = l'_v$ is unchanged by the definition of the parent relationship, since the only difference is v and v'.

Consider the shortest path between u, v (in $T_{u,v}$). By the distance algorithm, the next node in the path from v is $\texttt{parent}_{T_{u,v}}(v)$. Since the tree is unchanged for vertices w with $l_w < l_v = l'_v$, the shortest path between u, v' must coincide with the shortest path between u, v, since $\texttt{parent}_{T_{u,v}}(v) = \texttt{parent}_{T_{u,v'}}(v')$. Thus we have $\texttt{distance}_G(u, v) = \texttt{distance}_G(u, v')$.

Now consider the trees $T_{u,v'}$ and $T_{u',v'}$. In particular, the path to the root from v'. Let this path be $v' = p_1, \dots p_k = r_1$ in $T_{u,v'}$ and $v' = q_1, \dots, q_l = r_2$ in $T_{u',v'}$. By Lemma 4.4.2, the path from v' to u is found by the first index i where p_i intersects u. That is the first index i where $l_{p_i} \le r_u$. Similarly, the path from v' to u' is found by the first index j where q_j intersects u'. That is the first index j where $l_{q_j} \le r'_u$.

Now note that for all indices $h \le i, j$, $p_h = q_h$ (that is they represent the same vertex). To see this we induct on h. For $h = 1$, both p_1 and q_1 represent v'. For any $h < i, j$ $\texttt{parent}_{T_{u,v'}}(p_h) = \texttt{parent}_{T_{u',v'}}(q_h)$. This is because the set of candidate vertices in the definition of parent, given by equation 6.1 are the same. Neither u nor u' are candidates and that is the only difference between the two graphs.

Since $p_{\min(i,j)} = q_{\min(i,j)}$ and they both have the property that $l_{p_{\min(i,j)}} = l_{q_{\min(i,j)}} \le r_u = r_{u'}$, we see that indeed $i = j$ and $\texttt{distance}_G(u, v') = \texttt{distance}_G(u', v')$. □

Now we wish to treat $G \cup \{u', v'\}$ as a proper interval graph, so that we may use the machinery from the previous section. That is we wish to set the left end point of u' and right end point of v' so that it is consistent with the proper interval graph G.

Lemma 6.4.12. *Let G be a proper interval graph with intervals $\mathcal{I}$. Let r_x, l_y be two points such that r_x is not the right end point of any interval $I_v \in \mathcal{I}$ and l_y is not the left end point* *of any interval. Then there exists l_x, r_y such that $G \cup \{[l_x, r_x], [l_y, r_y]\}$ is a proper interval graph.*

Proof. For l_x, we find the successor u of r_x by $r_u = \texttt{succ}_R(r_x)$. We set the left end point so that it is proper by $l_{u'} = \texttt{pred}_L(l_u)$ and $l_x \in (l_{u'}, l_u)$. By doing so we guarantee the inequalities $l_{u'} < l_x < l_u$ and $r_{u'} < r_x < r_u$.

For r_y, we find the predecessor v of l_y by $l_v = \texttt{pred}_L(l_y)$. We ensure that it is proper by setting $r_y \in (r_v, r_{v'})$ where $r_{v'} = \texttt{succ}_R(r_v)$. □

Now we are able to prove the main theorem for this section. We are able to support the query of distances not only on vertices of proper interval graphs but on pairs of arbitrary vertices as well.

Theorem 6.4.13. *A proper interval graph G can be represented in $O(n)$ words of space, where n is the number of vertices currently in G, to support* `insert`, `delete`, `distance` *in $O(\lg n)$ time, and* `spath` *in $O(\lg n)$ time per vertex on the path. Furthermore,* `distance` *supports arguments not in the graph G: for intervals $x = [l_x, r_x], y = [l_y, r_y]$ not necessarily in G, $\texttt{distance}_{G \cup \{x,y\}}(x, y)$ is supported in $O(\lg n)$ time.*

Proof. First assume that $l_x < l_y$. If $r_x \geq l_y$ they are adjacent and we return 1.

We add the intervals $[l'_x, r_x], [l_y, r'_y]$ to G, query the distance, then delete the intervals, where l'_x, r'_y are found as in Lemma 6.4.12. If $r_x = r_v$ for some interval v, then we set $l'_x = l_v$ and we do not needed to add this interval, and we query using v. Similarly for y. As this uses $O(1)$ predecessor and successor queries, insertions and deletions, this takes $O(\lg n)$ time. Finally, we have by Lemma 6.4.11, $\texttt{distance}_G(x, y) = \texttt{distance}_G([r_x, r_x], [l_y, l_y]) = \texttt{distance}_G([l'_x, r_x], [l_y, r'_y])$ □

6.5 Incremental/Decremental Interval Graph

In this section, we will consider problem in general interval graphs. However, we will restrict the updates so that it is either only incremental or decremental. In other words, we are only allowed `insert` or we are only allowed `delete`.

We do this by observing that any interval that is contained in some other interval can be removed without changing the length of shortest paths. By maintaining the set of remaining intervals, which we will say are *exposed*, we reduce the problem to the fully dynamic proper case in Section 6.4. In the incremental setting, once an interval becomes contained by another interval (that is, no longer exposed), it will remain so for the remaining operations; in the decremental setting, once an interval becomes not contained by another interval (that is, becomes exposed), it will remain so until it is deleted. Hence, each interval will be added and deleted from the proper interval graph at most once. Therefore the total number of updates to the proper interval graph data structure will be $O(n)$ and we will achieve good amortized query times.

Definition 6.5.1. For an interval graph G with intervals $\mathcal{I}$, we say that an interval/vertex $x \in \mathcal{I}$ is ***exposed*** if it is not contained by another interval of $\mathcal{I}$, and we let $\mathcal{I}^{\texttt{exposed}}(G)$ denote the set of all exposed interval of G.

We note that by definition, the subgraph H of G consisting of exposed vertices form a proper interval graph. Let $x, y \in \mathcal{I}^{\texttt{exposed}}(G)$ be two exposed vertices, and by symmetry suppose that $l_x < l_y$. As x, y belong to a proper interval graph H, we also have $r_x < r_y$. As before, we will use $x < y$ to denote that $l_x < l_y$ for two exposed vertices.

As stated, we will reduce distance queries in G to distance queries in H.

Lemma 6.5.2. *Given an interval graph G with fixed interval representation $\mathcal{I}$, its exposed intervals $\mathcal{I}^{\texttt{exposed}}(G)$ form a proper interval graph H with the following properties:*

- *For any interval x, x is contained by an interval of G if and only if x is contained by an interval of H.*
- *For any two vertices $x, y \in G$, $\texttt{distance}_G(x, y) = \texttt{distance}_{H\cup\{x,y\}}(x, y)$*

Proof. If x is contained by an interval of H, x is contained by the same interval in G. Conversely, suppose x is contained by some other interval in G. Among intervals of G containing x, let y be the one with maximum length. Then, no other interval of G can contain y, or it would contain x and have longer length. Thus, $x \subseteq y \in H$.

Since $H \cup \{x, y\}$ is a subgraph of G, $\texttt{distance}_G(x, y) \le \texttt{distance}_{H\cup\{x,y\}}(x, y)$. Conversely, suppose $v_0 = x, v_1, v_2, \ldots, v_k = y$ is a shortest path between x and y in G. For each $1 \le i \le k-1$, we can replace v_i by a vertex of H containing it, and still have a path in $H \cup \{x, y\}$ of the same length. Hence, $\texttt{distance}_{H\cup\{x,y\}}(x, y) \le \texttt{distance}_G(x, y)$ □

Lemma 6.5.2 implies that to support $\texttt{distance}$ on interval graphs, it suffices to maintain an instance of fully dynamic proper interval graphs using Theorem 6.4.13, on the exposed vertices of G. Thus it remains to determine and maintain the exposed vertices of G. To do so, in addition to using an instance of the fully dynamic proper interval graphs data structure, we will also be storing the exposed intervals in an auxiliary data structure to help determine the changes.

Lemma 6.5.3. *There exists a data structure using $O(n)$ words that can store the exposed intervals and support the following operations:*

1. *Given an interval x, determine whether x is contained by some interval currently in the data structure, in $O(\lg n)$ time.*
2. *Given an interval x, report all intervals that are contained by x and delete all of them, in $O(\lg n + k)$ time, where k is the number of deleted intervals.*
3. *Insert an interval that does not contain and is not contained by any interval currently in the data structure, in $O(\lg n)$ time.*
4. *Given an interval in the data structure, return its predecessor with respect to $<$, or report that it doesn't exist.*
5. *Given an interval in the data structure, return its successor with respect to $<$, or report that it doesn't exist.*

Here, n denotes the number of intervals currently in the data structure.

Proof of Lemma 6.5.3 . We use a balanced binary search tree (Lemma 6.2.3) of the intervals using the left end points as the keys, and store the right end points as the value. To check if x is contained by some interval, it suffices to check the last interval whose left endpoint is at most l_x using Lemma 6.4.4. To check if x contains some interval, we just need to check the first interval whose left endpoint is at least l_x.

To find all such contained intervals we find the last interval y where $r_y \leq r_x$. Since the right endpoints are sorted in the same order as the left endpoints, we may binary search for the node in the tree. All nodes between the successor s of x and y are contained by x. To see this, let w be such a node. Then $s \leq w \leq y$, so that $l_x \leq l_s \leq l_w$ and $r_w \leq r_y \leq r_x$. Any vertices w to the left of s and the right of y by our choice would have either $l_w < l_x$ (if it were to the left of s) or $r_w > r_x$ (if it were to the right of y.

Since the nodes covered form a sequence of nodes in an in-order traversal, we find find them in $O(\lg n + k)$ time. By Lemma 6.2.3, we may delete each of them in $O(1)$ time each after we have found them. Thus the total time to delete the contained intervals is $O(\lg n + k)$.

The other operations are standard. □

6.5.1 Online Data Structure for Incremental Interval Graphs

We will now consider the incremental case, where the updates to the graph are only insertions of vertices. To do so, we investigate the changes to the exposed vertices upon the insertions of a new vertex.

Lemma 6.5.4. *Suppose G is an interval graph with fixed interval representation and $G' = G \cup \{x\}$ for some interval x. Then, $\mathcal{I}^{\texttt{exposed}}(G') \subseteq \mathcal{I}^{\texttt{exposed}}(G) \cup \{x\}$.*

Proof. Let y be an exposed interval of G'. Then, y is not contained by any other interval of $G \cup \{x\}$. If $y \in G$, then y must be exposed in G as well. □

Theorem 6.5.5. *There is a data structure that maintains an interval graph G in $O(n)$ words of space and supports the operations:*

1. *Insert a vertex represented by interval $[l, r]$ into G in $O(\lg n)$ amortized time,*
2. *Compute the distance between two arbitrary vertices x and y in G in $O(\lg n)$ worst-case time.*

Proof. We maintain an instance of the fully dynamic proper interval graph data structure described in Theorem 6.4.13 on the graph H whose vertices are the exposed intervals of our current interval graph G. We also maintain an instance of the binary search tree in Lemma 6.5.3 on the same intervals.

By Lemma 6.5.2, $\texttt{distance}_G(x, y)$ can be computed in $O(\lg n)$ time by querying $\texttt{distance}_{H\cup\{x,y\}}(x, y)$.

To handle insertion of x: if x is contained by some interval of G, we do nothing. Otherwise, we remove from H all intervals that are contained by x, and then insert x.

By Lemma 6.5.2, we can test if x is contained in some interval of G by testing if x is contained in some interval of H, which takes $O(\lg n)$ using Lemma 6.5.3. Removing k intervals from H takes $O(\lg n + k)$ time for the BST and $O(k \lg n)$ time for Theorem 6.4.13.

Since the total number of insertions to H is bounded by n, so must the total number of deletions. Hence, n `inserts` take $O(n \lg n)$ time total. □

6.5.2 Online Data Structure for Decremental Interval Graphs

In this subsection, we consider the decremental case, where the updates are the deletion of intervals. As in the incremental case, first, we investigate how the set of exposed intervals change after a deletion.

Lemma 6.5.6. *Let G be a proper interval graph, and $G' = G - x$ for some vertex $x \in G$. If x not exposed in G, then, $\mathcal{I}^{\text{exposed}}(G) = \mathcal{I}^{\text{exposed}}(G')$. If x is exposed in G, then $\mathcal{I}^{\text{exposed}}(G) \setminus \{x\} \subseteq \mathcal{I}^{\text{exposed}}(G')$. Furthermore, for all $y \in \mathcal{I}^{\text{exposed}}(G') \setminus \mathcal{I}^{\text{exposed}}(G)$, $y \subseteq x$.*

Therefore, if x^- is the predecessor of x in $\mathcal{I}^{\text{exposed}}(G)$ and x^+ is the successor of x in $\mathcal{I}^{\text{exposed}}(G)$, we have $\mathcal{I}^{\text{exposed}}(G') = (\mathcal{I}^{\text{exposed}}(G) \setminus \{x\}) \dot\cup \{y \in \mathcal{I}^{\text{exposed}}(G') : x^- < y < x^+\}$; that is the newly added elements of $\mathcal{I}^{\text{exposed}}(G')$ are between x^- and x^+.

Note: if either x^- or x^+ does not exist, then that constraint is omitted.

Proof. First, we show that $\mathcal{I}^{\text{exposed}}(G) \setminus \{x\} \subseteq \mathcal{I}^{\text{exposed}}(G')$. Let $y \in \mathcal{I}^{\text{exposed}}(G) \setminus \{x\}$. Then, $y \in G'$. Suppose for contradiction that y is contained by $z \in G'$. Then, y is contained by $z \in G$, contradicting $y \in \mathcal{I}^{\text{exposed}}(G)$.

In the case that x is not exposed, we have $\mathcal{I}^{\text{exposed}}(G) \subseteq \mathcal{I}^{\text{exposed}}(G')$. For the other direction, let $y \in \mathcal{I}^{\text{exposed}}(G')$. Then y is not contained by any interval of $G' = G - x$. If y were contained by x, then as x is not exposed, by Lemma 6.5.2 there exists an interval z containing x, which would also contain y. As $z \in G - x = G'$, y would not be exposed in G', a contradiction. Thus y is not contained by x so $y \in \mathcal{I}^{\text{exposed}}(G)$.

Now, suppose x is exposed. Let $y \in \mathcal{I}^{\text{exposed}}(G') \setminus \mathcal{I}^{\text{exposed}}(G)$. Since $y \notin \mathcal{I}^{\text{exposed}}(G)$, it must be contained by some interval $z \in G$. Since $y \in \mathcal{I}^{\text{exposed}}(G')$, $z \notin G'$. Thus, $y \subseteq x$.

By our choice of x^-, $l_{x^-} < l_x \leq l_y$. We cannot have $r_y \leq r_{x^-}$ as then $y \subseteq x^-$, contradicting $y \in \mathcal{I}^{\text{exposed}}(G')$. Hence, $x^- < y$. The proof that $y < x^+$ for $x < x^+$ is symmetric.

Lastly we note that by our choice of x^- and x^+, no element of $\mathcal{I}^{\text{exposed}}(G) \setminus \{x\}$ is between x^- and x^+ and thus $(\mathcal{I}^{\text{exposed}}(G) \setminus \{x\}) \cap \{y \in \mathcal{I}^{\text{exposed}}(G') : x^- < y < x^+\} = \emptyset$, so that the union is a disjoint union. □

We will now assume that the deleted vertex x is exposed, as there is nothing to be done if not. We wish to find set $\{y \in \mathcal{I}^{\text{exposed}}(G') : x^- < y < x^+\}$. To do so, we will iteratively find the exposed intervals y from smallest to largest.

Lemma 6.5.7. *Suppose we have a set of intervals S and let $x \in S$ be an exposed interval. Let $y \in S$ be the interval with minimum l_y (ties broken by largest r_y) such that $r_y > r_x$. Then, y is exposed, and there does not exist any exposed interval $z \in S$ such that $x < z < y$. If no such y exists, then there is no exposed interval w such that $x < w$.*

Proof. Suppose that w contains y: $y \subsetneq w \in S$. Then, $r_w \geq r_y > r_x$ and $l_w \leq l_y$, contradicting choice of y. Hence, y is exposed.

Next suppose that $x < z < y$. Then, $r_z > r_x$ (since z is exposed) and $l_z < l_y$, contradicting choice of y. Hence, no such z exists.

If $x < w$, then $r_w > r_x$, so w would be a candidate for y. □

Now that we know the criteria by which the exposed intervals can be found, it remains to construct the data structure to do so.

Lemma 6.5.8. *There exists a data structure using $O(n)$ words of space that can maintain a set of intervals (initially a given set) and do the following operations*

1. *Delete an interval in $O(\lg n)$*
2. *Given any two exposed intervals x and y where $x < y$, report all exposed intervals z such that $x < z < y$ in $O((k+1)\lg n)$, where k is the number of returned intervals. We also allow $x = -\infty$ or $y = \infty$ or both, in which case the constraint involving them is omitted.*

Proof. We store the intervals in a balanced binary search tree, where the keys are the right endpoints of intervals (ties are broken by left end points of intervals in reverse order), where at each node, we store the minimum left endpoint of all the intervals in the subtree.

To delete an interval, we delete it form the balanced binary search tree in $O(\lg n)$ time.

Given a exposed interval x, we can find the exposed interval y such that $x < y$ and there is no other exposed interval between them in the ordering $<$, or determine that none exists, by doing the following: query the BST to find the interval $[l_y, r_y]$ with minimum l such that $r_y > r_x$. This is our desired interval by Lemma 6.5.7. Note that if $x = -\infty$ we return the interval y with minimal l_y and maximal r_y among those tied with minimal l_y.

To perform the second operation, we use the following recursive algorithm

- Find exposed z such that $x < z$ and no other exposed interval is between them.

- If $z = y$, stop; otherwise, recursively find all exposed intervals between z and y, and return them together with z.

This finds all exposed intervals between x and y because if w is a exposed interval such that $x < w < y$, then either $w = z$, or $z < w < y$. □

Combining all the above results, we have the analogous theorem for the decremental case.

Theorem 6.5.9. *There is a data structure that starts with a given interval graph G, and supports using $O(n)$ words of space the operations:*

1. *Delete a vertex represented by interval $[l, r]$ from G in $O(\lg n)$ amortized time.*
2. *Compute the distance between two arbitrary vertices x and y in G in $O(\lg n)$ worst-case time.*

Proof. We maintain an instance of the fully dynamic proper interval graph data structure described in Theorem 6.4.13 on the graph H whose vertices are the exposed intervals of our current interval graph G. We also maintain an instance of the binary search tree in Lemma 6.5.3 on the exposed intervals, and an instance of the binary search tree in Lemma 6.5.8 on all the intervals of G.

By Lemma 6.5.2, $\texttt{distance}_G(x, y)$ can be computed in $O(\lg n)$ time by querying $\texttt{distance}_H(x, y)$.

When deleting an interval x:

- First, remove x from the BST of all intervals. If x is not exposed, stop.
- Now, suppose x is exposed. Let x^- and x^+ be the preceding and succeeding exposed interval, respectively. These can be found in $O(\lg n)$ using the BST of Lemma 6.5.3. Remove x from the proper interval graph data structure. Then, query the BST of all intervals (Lemma 6.5.8) to obtain all exposed intervals between x^- and x^+. Insert these into the the BST of exposed intervals, and into the proper interval graph data structure.

Correctness follows from Lemma 6.5.6. Since the total number of deletions to H is bounded by n, and the size of H never exceeds n, so must the total number of insertions. Hence, n `deletes` take $O(n \lg n)$ time total. □

6.6 Offline Dynamic Interval Graph

In this section, we will study the case when the operations are offline. That is, we are given the sequence of queries, $q_1, \ldots, q_t$ in advance where each q_i is either `insert`, `delete` or `distance`. Our goal is to return the result of all `distance` queries, in time $\widetilde{O}(n+t)$, where n is the size of the graph before hand, and t is the number of queries.

To do so, we will employ divide and conquer on time. Given a time interval $[t_1, t_2]$ (corresponding to queries $q_{t_1}, \ldots, q_{t_2}$), we will divide it into two smaller intervals $[t_1, (t_1 + t_2)/2]$ and $((t_1 + t_2)/2, t_2]$, and continue in this fashion until there is only a single query in the interval, then compute the result of the query. In order for the time complexity to arise, the amount of work needed to divide the time interval must be proportional to the length of the time interval.

6.6.1 Divide and Conquer on Time

Now we will describe the process for computing the queries using divide and conquer on time.

Definition 6.6.1. Let G be an interval graph with a fixed interval representation $\mathcal{I}$, and let S be a set of points. We will say that $R(G, S)$ is an ***reduced representation*** of G if:

- for any two intervals $u = [l_u, r_u], v = [l_v, r_v]$ with endpoints in S, we may compute the query
$$\texttt{distance}_{G\cup\{u,v\}}(u, v)$$
using $R(G, S)$
- If $\mathcal{I}_S$ is any set of intervals with endpoints in S, then we are able to compute $R(G', S)$ from $R(G, S)$ only (that is without requiring knowledge of G) for $G' = G \cup \mathcal{I}_S$. We will define this operation as $R(G', S) = \texttt{batch\_insert}(R(G, S), \mathcal{I}_S)$.
- If $S' \subset S$, then we are able to compute $R(G, S')$ from $R(G, S)$. We will define this operation as $R(G, S') = \texttt{reduce\_active}(R(G, S), S')$.

For an interval of queries $[t_1, t_2]$, we will say that any vertex of the graph G right before the query q_{t_1} that does not appear in any `insert` or `delete` queries in $q_{t_1}, \ldots, q_{t_2}$ as ***permanent***. Thus we may view the updates as small changes on a large permanent graph. Now consider reducing the interval of queries from $[t_1, t_2]$ to $[t_1, t_3]$ and $[t_4, t_2]$ where

$t_4 = t_3 + 1$. For $[t_1, t_3]$, any vertex that would be added in $[t_4, t_2]$ does not matter, since it would not affect any queries in the time interval $[t_1, t_3]$. Any vertex that would be deleted during $[t_4, t_2]$ (but not inserted/deleted in $[t_1, t_3]$) however would now become permanent, as they are a vertex in G but now are not part of any updates in $[t_1, t_3]$.

The opposite is true when we consider reducing to the interval $[t_2, t_4]$. Any deletions that occur in $[t_1, t_3]$ can be ignored, but any insertions that occur in $[t_1, t_3]$ must be added.

Thus for an interval graph G and a sequence of queries $q_{t_1}, \dots, q_{t_2}$, we will define $S(t_1, t_2)$ as the set of all endpoints of intervals that occur in some query in the sequence. We will let $P(G, t_1, t_2)$ denote the graph on the permanent vertices of G (which is the graph right before the query q_{t_1}) with respect to $[t_1, t_2]$. Let $R(P(G, t_1, t_2), S(t_1, t_2))$ be a reduced representation of $P(G)$. Then when we split the time interval, we will obtain two new graphs $P(G, t_1, t_3)$ and $P(G, t_4, t_2)$, and two new sets $S(t_1, t_3)$ and $S(t_4, t_2)$ with the property that $P(G, t_1, t_2) \subseteq P(G, t_1, t_3)$ and $P(G, t_1, t_2) \subseteq P(G, t_4, t_2)$ with $S(t_1, t_3) \subseteq S(t_1, t_2)$, and $S(t_4, t_2) \subseteq S(t_1, t_2)$.

By the properties of a reduced representation, we may obtain all of $R(P(G, t_1, t_3), S(t_1, t_3))$, $R(P(G, t_4, t_2), S(t_4, t_2))$ from $R(P(G, t_1, t_2), S(t_4, t_2))$. Our base case is when we reduce the time period $[t_1, t_1]$ so that there is a single query. If that single query is an update, we can ignore it, but if that single query is `distance`, then since we have $G = P(G, t_1, t_1)$, the query answered using $R(P(G, t_1, t_1), S(t_1, t_1))$ will be correct.

The above can be summarized in Algorithm 1.

The correctness of the algorithm follows from the properties of a reduced representation of G. Now consider the run time. Let $k = t_2 - t_1 + 1$ be the number of queries. Suppose that `batch_insert` takes time $O(k \lg^c k)$ for $c \geq 1$, that `reduce_active` takes time $O(k \lg^c k)$ and that `distance` takes $O(1)$ time, then Algorithm 1 has the recurrence

$$T(k) = 2T(k/2) + O(k \lg^c k)$$

This solves to a run time of $T(n) = O(n \lg^{c+1} n)$.

It now remains to construct a reduced representation of G.

6.6.2 Reduced Representation of an Interval Graph

One obvious candidate for the reduced representation of an interval graph is its distance tree Definition 6.2.1. However, to be able to support `batch_insert` and `reduce_active` in time proportional to the number of queries (i.e. $k = t_2 - t_1 + 1 = O(|S(t_1, t_2)|)$) we cannot build the tree on all the vertices of G which does not have size proportional to $|S|$.

```
   input : Queries $q_{t_1}, \dots, q_{t_2}$, Reduced representation $R(P(G, t_1, t_2), S(t_1, t_2))$ and
           $S(t_1, t_2)$
   output: The result of all the distance queries in $q_{t_1}, \dots, q_{t_2}$
 1 Function ComputeQueries;
 2 if $t_1 = t_2$ then
 3     if $q_{t_1} =$ distance$(u, v)$ then
 4         return distance$_{R(P(G,t_1,t_1),S(t_1,t_1))}(u, v)$
 5     end
 6 end
 7 $t_3 \longleftarrow \lfloor (t_1 + t_2)/2 \rfloor$;
 8 Compute $\mathcal{I}_S^1$ as the vertices removed in $q_{t_3+1}, \dots, t_2$ but not inserted in $q_{t_1}, \dots, q_{t_3}$;
 9 Compute $\mathcal{I}_S^2$ as the vertices inserted in $q_{t_1}, \dots q_{t_3}$ but not removed in $q_{t_3+1}, \dots, q_{t_2}$;
10 Compute $R(P(G, t_1, t_3), S(t_1, t_2)) =$ batch_insert$(R(P(G, t_1, t_2), S(t_1, t_2)), \mathcal{I}_S^1)$;
11 Compute
     $R(P(G, t_3 + 1, t_2), S(t_1, t_2)) =$ batch_insert$(R(P(G, t_1, t_2), S(t_1, t_2)), \mathcal{I}_S^2)$;
12 Compute $S(t_1, t_3)$ and $S(t_3 + 1, t_2)$;
13 Compute
     $R(P(G, t_1, t_3), S(t_1, t_3)) =$ reduce_active$(R(P(G, t_1, t_3), S(t_1, t_2)), S(t_1, t_3))$;
14 Compute $R(P(G, t_3 + 1, t_2), S(t_3 + 1, t_2)) =$
     reduce_active$(R(P(G, t_3 + 1, t_2), S(t_1, t_2)), S(t_3 + 1, t_2)))$;
15 return ComputeQueries $(q_{t_1}, \dots, q_{t_3}$, $R(P(G, t_1, t_3), S(t_1, t_3))$, $S(t_1, t_3))$;
16 return ComputeQueries $(q_{t_3+1}, \dots, q_{t_2}$, $R(P(G, t_3 + 1, t_2), S(t_3 + 1, t_2))$,
     $S(t_3 + 1, t_2))$
```
Algorithm 1: Algorithm for divide and conquer on time

However, we note that for the computation of distance, we only need the path to the root. Since all of our queries must come from vertices with end points in S, only the path to the root from nodes that would represent endpoints in S would matter. All other nodes in the tree would never be needed for any queries, and can be removed. Thus we need to store the points of S which are endpoints of possible queries in the tree as well, since their path to the root is what is important. Therefore in distance tree, we may remove all the non-necessary nodes.

First we note that by Lemma 6.5.2 it suffices to maintain a proper interval graph. Furthermore, by Lemma 6.4.11, given two vertices u, v with $l_u < l_v$, their distance only depends on r_u and l_v, and in Theorem 6.4.13, we add intervals that is consistent with the interval graph with those end points. However, we note that all nodes in the distance tree corresponds to the left endpoint of some interval, and r_u would be the right endpoint of some interval. We first remove this inconsistency.

Lemma 6.6.2. *Let $\hat{G}$ be a proper interval graph with intervals $\mathcal{I}$. Let $u' = [r_u, r_u]$ and $v' = [l_v, l_v]$ with $r_u < l_v$. Let $\hat{u} = [r_u, r_{\hat{u}}]$ ($l_{\hat{u}} = r_u$) be an interval where $r_{\hat{u}}$ is chosen so that it is consistent with the intervals $\mathcal{I}$ and that it has no children. Then* $\texttt{distance}(u', v') = \texttt{distance}(\hat{u}, v') + 1$.

Proof. Consider the path to the root from v' as $v' = p_1, \dots, p_k = r$. By Lemma 4.4.2 the shortest path is found by picking the first index i such that $l_{p_i} \le r_u$.

By minimality of i, we have $l_{p_i} < r_u < l_{p_{i-1}}$. Now consider the interval for $\hat{u}$. We claim that $l_{p_{i-1}} < r_{\hat{u}}$ is the first index where this is true.

First we show that $l_{p_{i-1}} \le r_{\hat{u}}$. By the definition of parent we have $l_{p_{i-1}} \le r_{p_i}$, thus we have $l_{p_i} < r_u = l_{\hat{u}} < l_{p_{i-1}} \le r_{p_i} < r_{\hat{u}}$.

Now suppose that $l_{p_{i-2}} \le r_{\hat{u}}$. By definition of parent, we have $l_{p_{i-2}} \le r_{p_{i-1}} \le r_{\hat{u}}$, and so $l_{p_{i-1}} \le l_{\hat{u}} = r_u < l_{p_{i-2}} \le r_{p_{i-1}} \le r_{\hat{u}}$, contradicting the minimality of i. □

The above lemma shows that we may treat r_u as the left endpoint of some interval, and compute the distance using that. Now we show that we may always simply put the point r_u into the distance tree as a leaf without affecting the rest of the tree structure.

Lemma 6.6.3. *Let $\hat{G}$ be a proper interval graph with intervals $\mathcal{I}$. Let l_u be a point that is not the left endpoint of any interval $I \in \mathcal{I}$, then there exists a right end point r_u such that $[l_u, r_u]$ is consistent with $\mathcal{I}$ and the vertex $u = [l_u, r_u]$ has no children in the distance tree if it were added to the graph.*

Proof. Let v, v' be the predecessor and successor of u respectively in G. That is $l_v < l_u < l_{v'}$ are the nearest left endpoints to l_u. In order for r_u to be consistent, we must have $r_v < r_u < r_{v'}$. Consider the left endpoints of vertices between r_v and $r_{v'}$. Let $v_1, \ldots, v_k$ be these vertices so that $r_v < l_{v_1} < \ldots < l_{v_k} \leq r_{v'}$. We note that by the definition of $\texttt{parent}_{T(G)}$, all of these vertices have v' as their parent. Choose $r_u \in (r_v, l_{v_1})$. This ensures that u is not the parent of any vertex in G. $\square$

Thus given the distance tree of the proper interval graph on the exposed vertices $\mathcal{I}^{\texttt{exposed}}(G)$, and the set of points S, we may construct an augmented distance tree which includes the points of S.

Definition 6.6.4. Let G be an interval graph and $\hat{G}$ be the proper interval graph, with intervals $\mathcal{I}$. Let S be a set of points. The graph $(\hat{G}, S)$ is the graph obtained by treating each point in S as the left end point of an interval (if it does not exist as an interval of $\hat{G}$ already) with the left end point being the point in S, and the right endpoint chosen so that it is consistent with the intervals of $\hat{G}$ and that it has no children.

The main thing we note in this definition is that if $\hat{G}$ changes, then the interval assigned to a point in S may no longer be consistent. However, since we do not need to assign an interval to the points in S (as they are placeholders to represent *some consistent interval*) we may always find a new consistent interval for that point. Since the only thing that is important for these points is their parent, when $\hat{G}$ changes, the only thing that matters is to confirm the new parent of a point in S.

Definition 6.6.5. For the distance tree $T_{(\hat{G},S)}$, we say that a vertex is ***active*** if it is the vertex corresponding to a point in S or it is a vertex v such that $l_v \in S$. (Note that by Definition 6.6.4, we insert all points in S that is not the left end point of any interval in $\hat{G}$)

Now we will compress this tree so that updates to it can be done in time proportional to $|S|$, which is also the number of active vertices.

Definition 6.6.6. Define the ***compressed distance tree*** $\hat{T}_{(\hat{G},S)}$ by compressing all paths in $T_{(\hat{G},S)}$ by keeping only active vertices and any lowest common ancestors of active vertices. $\hat{T}$ is a weighted tree where the edges are weighted by the lengths of the compressed paths.

It is clear that the number of vertices in $\hat{T}$ is $O(|S|)$. Now we show that $\hat{T}$ is sufficient to compute distances.

Lemma 6.6.7. *Consider the proper interval graph $(\hat{G}, S)$. Let u, v be two vertices such that $l_u, r_u, l_v, r_v \in S$, Then we may compute* $\texttt{distance}_(\hat{G})(u, v)$ *using* $\hat{T}_{(\hat{G},S)}$.

Proof. First we note that all points of S are leaves in the tree and thus do not affect any shortest paths not involving them. Now suppose that $l_u < l_v$. If $r_u > l_v$, then we return 1 immediately, so assume that $r_u < l_v$. In the uncompressed tree $T_{(\hat{G},S)}$, we note that the distance is found by using the representatives of r_u, l_v (via Lemma 6.6.2 wihch states that we only need to compute the distance between these points and add 1).

By compressing the lengths of the paths, we do not change the distances, nor do we change the relative positioning of the two vertices (in post-order), and thus Lemma 6.2.2 still holds. □

As each node in $\hat{T}$ represents a compressed path, we will refer to this path as the compressed path of v.

Now we will describe our reduced representation for an interval graph G with respect to the queries $q_{t_1}, \ldots, q_{t_2}$. Let $S = S(t_1, t_2)$ be the endpoints of the queries. We take the proper interval graph $\hat{G}$ using the exposed intervals $\mathcal{I}^{\texttt{exposed}}(G)$. Construct the distance tree $\hat{T}_{(\hat{(G)},S)}$. It is clear that we may construct this in $O(n \lg n)$ time by traversing the tree where n is the number of vertices of G.

We now store some extra information about the nodes of $\hat{T}$. For every node $v \in \hat{T}$, we store the point $(\texttt{node\_rank}_{\texttt{POST}}(u), \texttt{depth}(u))$ in a dynamic 2-dimensional orthogonal range search data structure using Lemma 6.2.5

It now remains to show how to implement the operations `batch_insert` and `reduce_active` on our representation.

First we will handle the easier of the two operations: `reduce_active`.

Lemma 6.6.8. *Let $(\hat{G}, S)$ be a proper interval graph on vertices and points. Let $S_1 \subseteq S$. Then we can compute $\hat{T}_{(\hat{G},S\setminus S_1)}$ from $\hat{T}_{(\hat{G},S)}$ in $O(|S| \lg |S|)$ time. We may also compute the range search data structure on $\hat{T}_{(\hat{G},S\setminus S_1)}$ in $O(|S| \lg |S|)$ time. Hence we may support* `reduce_active` *in $O(|S| \lg |S|)$ time.*

Proof. By removing S_1 from S, we need to relabel nodes in the tree as active. Afterwards, we need to compress the tree again onto the smaller set of terminals. Both of these operations takes $O(|S| \lg |S|)$ time as the tree has size $O(|S|)$.

Lastly, we perform a post-order traversal of the tree and add the points we obtain to the range search data structure. The traversal and the construction of the range search data structure takes $O(|S| \lg |S|)$ time. □

Now we consider the `batch_insert` operation. As shown in Lemma 6.5.4, the insertion of a vertex would delete many exposed vertices, which must be consecutive. Thus the modifications to $\hat{T}$ is the deletion of some vertices of $\hat{G}$ and the re-computation of the parent of some vertices.

In the first case, we will say that a vertex is deleted, in the second we will say that a vertex is re-parented. In either case, we will say that a vertex *participates* in the update. Now consider the case that we are dealing with the compressed tree $\hat{T}$. If a vertex in the compressed path of v is deleted, then some other vertex in the compress path will need to re-parent. We will say that if a vertex in the compressed path of v re-parents that v re-parents (since v will now have a new parent in $\hat{T}$ and it will carry its compressed path with it).

More specifically, in the `batch_insert` operation, we add a set of intervals with endpoints in S. Let $\mathcal{I}_{\mathcal{S}}$ be a set of at most $|S|$ intervals with endpoints in S. Define the graph $(G \cup \mathcal{I}_{\mathcal{S}}, S)$ be the graph obtained by adding the intervals in I_S to G. Let $(\hat{G} \cup \mathcal{I}_{\mathcal{S}}, S)$ be the proper interval graph obtained by considering the exposed vertices. We note that any non-exposed interval of G does not appear in $\hat{G}$ already and thus we may obtain $(\hat{G} \cup \mathcal{I}_{\mathcal{S}}, S)$ from $(\hat{G}, S)$.

Since we are only considering exposed intervals, the first step is to make sure that any two intervals we are adding $[l_u, r_u], [l_v, r_v]$ are not contained in each other, as the contained vertex will not appear in the proper interval graph $\hat{G} \cup \mathcal{I}_{\mathcal{S}}$. We may do this in $O(|S| \lg |S|)$ time since $|\mathcal{I}_S| = O(|S|)$. One way is to insert all of these vertices into the tree described in 6.5.3.

We will state results for T, and then translate those into results for $\hat{T}$. First we will begin with effect of a single insertion. Recall that by Lemma 4.4.7, for a vertex v of a proper interval graph, the right most neighbour of v is either the last child of v or if v is a leaf, the last child of the previous internal node of v in the distance tree. We will denote this node as `rightmost_neighbour`(v).

Lemma 6.6.9. *Let $(\hat{G}, S)$ be a proper interval graph, $T_{(\hat{G},S)}$ be the distance tree, and let $I_u = [l_u, r_u]$ be an interval with endpoints in S (we will refer to the nodes in the tree representing these terminals by l_u and r_u). Then we obtain $T_{(\hat{G} \cup \{I_u\}, S)}$ by the following (note that the set of nodes specified in each of the cases could be empty, and between means in a level-order traversal):*

- *Nodes v between l_u and (but not including)* `parent`(r_u) *are deleted - if any are active, their parent is now the earlier (in level order) between* `parent`(v) *and l_u.*

- *Nodes between the (righter most of* `parent`(r_u) *(this is included) and* `rightmost_neighbour` *(this is excluded)) and r_u will now have parent l_u (if they do not already).*

- *Furthermore, children of delete nodes are either deleted or will now have parent l_u. Therefore any active nodes that are deleted will not have any children.*

Proof. We are adding the interval I to the proper interval graph. As the result must be proper, the intervals v deleted are those with $l_u \leq l_v < r_v \leq r_u$. The intervals that satisfy the condition on l_v are the nodes in the tree between l_u and r_u. We claim the intervals that satisfy the condition on the right end point r_v are those to the left of `parent`(r_u).

Let v be a node in the tree to the left of `parent`(r_u). Then the right most neighbour of v, `rightmost_neighbour`(v) is to the left of r_u. Since v is not adjacent to the vertex with left endpoint r_u, we have $r_v < r_u$ and it should be deleted.

On the other hand, suppose that v is equal to or to the right of `parent`(r_u). Then `rightmost_neighbour`(v) is equal to or to the right of r_u and thus $r_v > r_u$ and v is not deleted.

If any of these deleted nodes are active, then after insertion, these nodes which represent the left end point of some consistent interval will have parent either l_u or the same parent as before. By Lemma 6.6.3, we may choose a right endpoint so that it is still consistent with the remaining intervals, and still have no children.

Now we consider the nodes which will have l_u as their parent in $T_{(\hat{G}\cup\{I_u\},S)}$. These intervals v must have $l_u \leq l_v \leq r_u < r_v$. Thus again, these are the vertices between l_u and r_u. Since $r_v > r_u$, by above, v must be equal to or to the right of `parent`(r_u). Let u' be the predecessor of l_u. Then we must have $l_{u'} < l_u \leq r_{u'} < l_v$, as other wise v would already have more leftward parent. Now consider `rightmost_neighbour`(l_u). If v equal to or to the left of this, then either: v has parent l_u already and nothing needs to be done, or v would have a parent to the left of l_u and nothing needs to be done. Conversely, if v is to the right of `rightmost_neighbour`(l_u), then as v is not adjacent to u, we have $l_v > r_u > r_{u'}$ as needed.

Lastly, suppose that v is deleted, then $l_u < l_v < r_v \leq r_u$. Let w be a child of v. Since w is adjacent to v, we have $l_w \leq r_v$ and hence $l_u < l_w \leq r_u$ and w satisfies one of the cases above. □

By our terminology, the nodes which satisfy the first point are deleted. Nodes which satisfy the second are re-parented. We note that even though active vertices satisfying the first point remain on the tree, we will still say that they are in the "deleted" category.

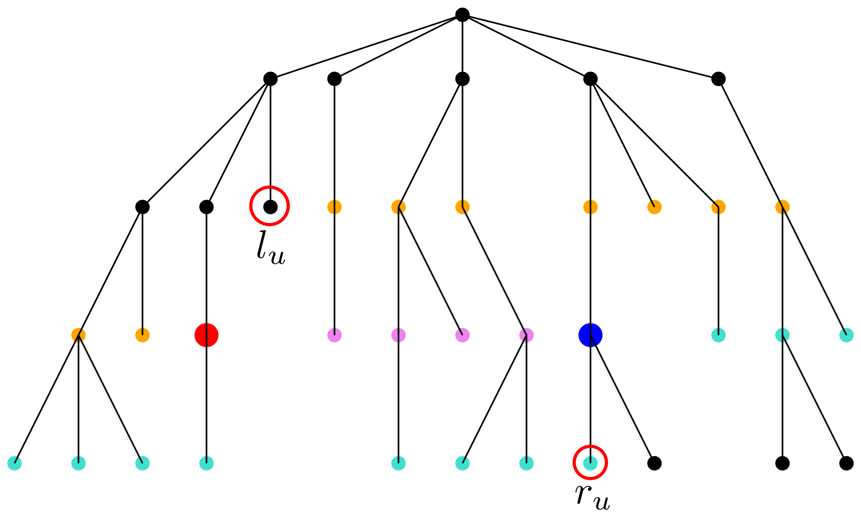


Figure 6.1: A distance tree undergoing a single insert. The nodes with different actions are coloured.

Example 6.6.10. Consider the tree in Figure 6.1. We have not depicted all the active nodes, except the two that are important: l_u and r_u corresponding to the vertex u we are inserting. The red node is $\texttt{rightmost\_neighbour}(l_u)$ the right-most neighbour of u, and the blue node is $\texttt{parent}(r_u)$. Thus the nodes between l_u and r_u can be split up into 3 categories: the orange nodes (including the red node in this case) are considered deleted, and if any are active, keep their parent, as their original parent is to the left of l_u. The magenta nodes are deleted, and if any are active will have l_u as their parent. The teal nodes re-parents and will have parent l_u. The following results simply restate this criteria using depths and post-order numbers so that it can be applied when the tree is compressed.

Now that we have stated the changes needed on the uncompressed tree $T_{(\hat{G},S)}$, we now need to consider the operation on the compressed tree. Since we do not have all the nodes, and most paths are compressed, we will rely on the only things that can be translated between the two trees: depths and relative orderings in a traversal. Thus this is essentially a restatement of Lemma 6.6.9.

Corollary 6.6.11. *Let $(\hat{G}, S)$ be a proper interval graph, $T_{(\hat{G},S)}$ be the distance tree, and let $I_u = [l_u, r_u]$ be an interval with endpoints in S (we will refer to the nodes in the tree representing these terminals by l_u and r_u). Then we obtain $T_{(\hat{G}\cup\{I_u\},S)}$ by the following (note that the set of nodes specified in each of the cases could be empty, and between means in a level-order traversal):*

- *The nodes deleted are those whose 2d point* ($\texttt{node\_rank}_{\texttt{POST}}, \texttt{depth}$) *satisfies at least one of the rectangles below:*

  Case 1: if $\texttt{depth}(l_u) = \texttt{depth}(\texttt{parent}(r_u))$*: no nodes if* $\texttt{node\_rank}_{\texttt{POST}}(r_u) \leq \texttt{node\_rank}_{\texttt{POST}}(l_u)$*, otherwise the nodes are in those in the rectangle*

$$(\texttt{node\_rank}_{\texttt{POST}}(l_u), \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(l_u), \texttt{depth}(l_u)]$$

  Case 2: The nodes between l_u *and* $\texttt{parent}(r_u)$ *are those in the rectangles*

$$(\texttt{node\_rank}_{\texttt{POST}}(l_u), \infty] \times [\texttt{depth}(l_u), \texttt{depth}(l_u)]$$

  plus

$$[-\infty, \infty] \times [\texttt{depth}(l_u) + 1, \texttt{depth}(r_u) - 2]$$

  plus

$$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$$

- *The nodes that will now have parent* l_u *are those whose 2d point satisfies one of the following rectangles:*

  Case 1: $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 1$*:*

$$(\texttt{node\_rank}_{\texttt{POST}}(l_u), \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$

  Case 2: $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 2$*:*

$$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$

  and

$$(\max\{\texttt{node\_rank}_{\texttt{POST}}(r_u), \texttt{node\_rank}_{\texttt{POST}}(l_u)\}, \infty] \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$$

  Case 3:

$$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$

  and

$$(\texttt{node\_rank}_{\texttt{POST}}(r_u), \infty] \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$$

Proof. First we note that if $\texttt{depth}(l_u) = \texttt{depth}(r_u)$, then there is nothing to be done as this guarantees the interval $[l_u, r_u]$ is contained in another.

For the vertices to be deleted, we break it up into two cases. In the first case, $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 1$ so that l_u has the same depth as $\texttt{parent}(r_u)$. If $\texttt{parent}(r_u)$ is to the left of l_u as indicated by their post order numbers, then there are no nodes to the right of l_u but to the left of $\texttt{parent}(r_u)$. Otherwise, the nodes deleted are at depth $\texttt{depth}(l_u)$ with post order numbers between the two.

Otherwise, we break the vertices between l_u and $\texttt{parent}(r_u)$ into a few cases. The ones on the same level as l_u, the ones on the same level as $\texttt{parent}(r_u)$ and those in between. These are given by the 3 rectangles.

Now for the nodes that will point to l_u, which are between $\texttt{parent}(r_u)$ and r_u, but also to the right of $\texttt{rightmost\_neighbour}(l_u)$. We again break it into a few cases:

First is $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 1$. In this case, we note that if $\texttt{rightmost\_neighbour}(l_u)$ has the same depth as r_u, then the nodes to its right must have post order numbers greater than l_u. which is captured by the rectangle given. In all cases, no nodes on the same depth as l_u will be to the right of $\texttt{rightmost\_neighbour}(l_u)$.

Next is $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 2$. Then all nodes on the same depth as r_u will be to the right of $\texttt{rightmost\_neighbour}(l_u)$. On the level above, the nodes to the right of $\texttt{parent}(l_u)$ and $\texttt{rightmost\_neighbour}(l_u)$ will have post order numbers greater than both of them, which is given by the rectangle.

Lastly, if $\texttt{depth}(l_u) < \texttt{depth}(r_u) - 2$, then $\texttt{parent}(r_u)$ is to the right of $\texttt{rightmost\_neighbour}(l_u)$ and so we do not need to consider $\texttt{rightmost\_neighbour}(l_u)$ at all. □

Now that we've restated the conditions in terms of depths and post order numbers, it becomes clear on how to do this on the compressed tree as it preserves depths and relative orderings of nodes in the post order traversal. The key insight now is that for a node v whose path is compressed, if one of the nodes on the compressed path of v satisfies a rectangle $[...] \times [a, b]$ if and only if the post order rank of v satisfies the first dimension, and its second coordinate satisfies $[a, \infty]$ and its parent's (in the compressed tree) depth is less than b. Therefore, for deletion, we will only consider the vertices (since if a node on the compressed path of v would be deleted, but v would not, there exists another node on the compressed path that would re-parent, which would find v to re-parent), but for re-parenting, we will consider both vertices and also compressed paths. By above, since deleted active vertices' children are considered in deletions and re-parenting, and have no children afterwards, we do not need to consider deleted active vertices' children.

Lemma 6.6.12. *Let $(\hat{G}, S)$ be a proper interval graph, $\hat{T}_{(\hat{G},S)}$ be the compressed distance tree, and let $I_u = [l_u, r_u]$ be an interval with endpoints in S (we will refer to the nodes in the tree representing these terminals by l_u and r_u). Then we obtain $\hat{T}_{(\hat{G}\cup\{I_u\},S)}$ by the following (note that the set of nodes specified in each of the cases could be empty, and between means in a level-order traversal):*

- *Note that any non-active nodes deleted implies that there is an active node descendant of it that may need to re-parent. Thus we only consider active node deletions, which causes the active node in this range to re-parent:*

  The active nodes deleted are those whose 2d point $(\texttt{node\_rank}_{\texttt{POST}}, \texttt{depth})$ *satisfies at least one of the rectangles below:*

  If $\texttt{depth}(l_u) < \texttt{depth}(r_u) - 1$*: The nodes that needs re-parenting are*

  $$(\texttt{node\_rank}_{\texttt{POST}}(l_u), \infty] \times [\texttt{depth}(l_u) + 1, \texttt{depth}(l_u) + 1]$$

  and

  $$[-\infty, \infty] \times [\texttt{depth}(l_u) + 2, \texttt{depth}(r_u) - 2]$$

  and

  $$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$$

  Will need to have parent l_u with weight 1, with children ordering based on depth, and if tied based on post order numbers.

- *For a vertex v satisfying one of the rectangles below (which means that for the second coordinate, if the one given is $[a, b]$, we use $[a, \infty]$), we also require that* $\texttt{parent}_{\hat{T}}(v) < b$.

  The nodes that will now have parent l_u are those whose 2d point satisfies one of the following rectangles:

  Case 1: $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 1$*:*

  $$(\texttt{node\_rank}_{\texttt{POST}}(l_u), \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$

  Case 2: $\texttt{depth}(l_u) = \texttt{depth}(r_u) - 2$*:*

  $$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$

  and

  $$(\max\{\texttt{node\_rank}_{\texttt{POST}}(r_u), \texttt{node\_rank}_{\texttt{POST}}(l_u)\}, \infty] \times [\,\texttt{depth}(r_u) - 1]$$

Case 3:
$$[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u), \texttt{depth}(r_u)]$$
and
$$(\texttt{node\_rank}_{\texttt{POST}}(r_u), \infty] \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$$

If a point satisfies more than one rectangle, the rectangle that is satisfies is the one with the largest value in the second dimension. The new weight of vertex is based on the distance formula of Lemma 6.2.2: let v be the vertex and suppose that the rectangle that is satisfies is $[...] \times [a, b]$*, the weight is* $\texttt{depth}(v) - \texttt{depth}(r_u) + \mathbf{1}(\texttt{node\_rank}_{\texttt{POST}}(v) > \texttt{node\_rank}_{\texttt{POST}}(r_u))$*.*

Proof. We note that the condition that an active vertex v satisfying $[...] \times [a, \infty]$ and $\texttt{depth}(\texttt{parent}_{\hat{T}}(v)) < b$ implies that the compressed path between v and its parent passes through the rectangle and thus one of the vertices on that path satisfies the rectangle.

Now we consider the cases: as noted, for the deleted vertices, if they have a descendant, then one vertex on that path will have l_u as its parent, thus we only consider those in the rectangles that are active, and for these, we do not extend the rectangles.

In Corollary 6.6.11, we had two cases: Case 1: if $\texttt{depth}(l_u) = \texttt{depth}(\texttt{parent}(r_u))$: no nodes if $\texttt{node\_rank}_{\texttt{POST}}(r_u) \leq \texttt{node\_rank}_{\texttt{POST}}(l_u)$, otherwise the nodes are in those in the rectangle $(\texttt{node\_rank}_{\texttt{POST}}(l_u), \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(l_u), \texttt{depth}(l_u)]$.

Case 2: The nodes between l_u and $\texttt{parent}(r_u)$ are those in the rectangles $(\texttt{node\_rank}_{\texttt{POST}}(l_u), \infty] \times [\texttt{depth}(l_u), \texttt{depth}(l_u)]$ plus $[-\infty, \infty] \times [\texttt{depth}(l_u)+1, \texttt{depth}(r_u)-2]$ plus $[-\infty, \texttt{node\_rank}_{\texttt{POST}}(r_u)) \times [\texttt{depth}(r_u) - 1, \texttt{depth}(r_u) - 1]$.

In the first case, all active vertices satisfying the rectangle are on the same level as l_u and thus are to the left of $\texttt{rightmost\_neighbour}(l_u)$. Hence their parents will not change upon the addition of I_u. In the second case, the first rectangle has the same property. The second rectangle we split into those on the level directly below l_u. The vertices there to the left of $\texttt{node\_rank}_{\texttt{POST}}(l_u)$ are to the left of $\texttt{rightmost\_neighbour}(l_u)$ and does not change their parent. Those to the right now has l_u as parent. The rest of the vertices in the remaining rectangles will also have l_u as parent since they are now adjacent to l_u and their original parent is to the right of l_u.

Now for the second step. Any active vertex whose path contains a vertex in the rectangle will re-parent. Note that all such rectangles are of the form $[a, a]$ and thus the vertex on the path will have depth a. The length of the path to this vertex is $\texttt{depth}(v) - a$. In all cases, $a = \texttt{depth}(r_u)$ when $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(r_u)$ and $a = \texttt{depth}(r_u) - 1$

when $\texttt{node\_rank}_{\texttt{POST}}(v) > \texttt{node\_rank}_{\texttt{POST}}(r_u)$. Thus the weight is $\texttt{depth}(v) - \texttt{depth}(r_u) + \mathbf{1}(\texttt{node\_rank}_{\texttt{POST}}(v) > \texttt{node\_rank}_{\texttt{POST}}(r_u))$. □

As we are implementing `batch_insert`, we will now consider the case of adding multiple vertices at once. In order to keep the total run time $O(|S| \lg |S|)$, we will ensure that each vertex will be returned by the range reporting data structure a constant number of times among the insertions, thus we will need to be careful on the vertices returned. We will again first consider the uncompressed tree T.

Lemma 6.6.13. *Let $(\hat{G}, S)$ be a proper interval graph with distance tree $T_{(\hat{G},S)}$ and compressed distance tree $\hat{T}_{(\hat{G},S)}$. Let $\mathcal{I}_S = \{I_{u_i} \mid i = 1 \dots k\}$ be a set of at most $|S|$ intervals with endpoints in S such that for every i, $l_{u_i} < l_{u_i+1} \le r_{u_i} < r_{u_{i+1}}$ (that is these intervals form a connected proper interval graph). Furthermore, suppose that for every interval I_{u_i}, no interval of $\hat{G}$ contains it (otherwise the insertion of this interval has no effect). Then the effect of inserting these intervals is as follows:*

- *Only vertices v in T (and thus either in $\hat{T}$ explicitly or is part of a compressed path) affected are those with $l_{u_1} < l_v \le r_{u_k}$*
- *A vertex v is deleted if there exists i (and we pick i to be the largest) such that $l_{u_i} \le l_v < r_v \le r_{u_i}$. Furthermore, If v is active, then let j be the first index such that $l_v \le r_j$, then the result of v after the insertion of I_{u_l} for all $l \ge j$ is the same as only inserting I_{u_j}.*
- *Otherwise for a vertex v, there exists an index i such that $r_{u_{i-1}} < l_v \le r_{u_i}$. The result of v after the insertion of I_{u_l} for all $l \ge i$ is the same as the insertion of I_{u_i}. (Here we allow $i = 1$ indicating that $l_v \le r_{u_1}$)*

Proof. The first point is clear. Only the vertices in the range are affect by the insertion of $\mathcal{I}_S$ as otherwise, they're parents will definitely not change after the insertion, nor will they be deleted.

Now consider a vertex v and an index i such that $l_{u_i} \le l_v < r_v \le r_{u_i}$. Then v is contained in I_{u_i} and thus would be removed after the insertion of u_i. However, if v is active, then it only represents the left end point of an interval, and by Lemma 6.6.3 we may choose the right end point so that it is consistent with the intervals, thus it must stay.

Now consider its parent. Let j be as defined. Since r_{u_j} is the first right end point, it is the best candidate to be the parent of v among $\mathcal{I}_S$. As $l_v \le r_{u_j}$, v will be considered in one of the cases when I_{u_j} is inserted. Now suppose that p the parent of v remains the same, that is $r_p < r_{u_j}$, then $r_p < r_{u_l}$ for $l > j$ and thus its parent will remain the same for the insertion of all other I_{u_l} for $l > j$. Otherwise, upon the insertion of u_j, v will have parent u_j. In both these cases, the effect of insertion I_{u_l} for $l > j$ have no effect on the final placement of v.

Lastly, we assume that no inserted interval contains v. Then there exists an index i such that $r_{u_{i-1}} < l_v \leq r_{u_i}$. By the same argument, if after inserting u_i, v has it same parent, then the insertion of all other I_{u_l} with $l > i$ will also keep the parent, and if after inserting u_i, the parent of v is u_i. Thus the effect is the same as if we only inserted u_i. □

Now we will consider the effect on the compressed tree. By doing so, we will have implemented `batch_insert` as we can construct the new range reporting data structure by traversing the new $\hat{T}$.

Lemma 6.6.14. *$(\hat{G}, S)$ be a proper interval graph with compressed distance tree $\hat{T}_{(\hat{G},S)}$. Let $\mathcal{I}_S$ be a set of at most $|S|$ intervals with endpoints in S. Then we can compute $\hat{T}_{(\hat{G}\cup\mathcal{I}_S,S)}$ in $O(|S| \lg |S|)$ time.*

Proof. First remove all intervals of $\mathcal{I}_S$ that are contained in another interval of $\mathcal{I}_S$ as they would not appear in the resulting graph. Next remove all intervals of $\mathcal{I}_S$ that is contained in some interval of G. We detect this my constructing the rectangles for that interval and checking if all of them are trivially empty (i.e one dimension is $[a, b]$ with $a > b$).

The remaining intervals form a proper interval graph. We insert each component of this proper interval graph separately. Starting from the right most component, we insert the intervals of each component starting from the right. Let $u_1, \ldots u_k$ be the vertices of the component to be added, named from left to right.

Applying Lemma 6.6.13 we insert u_k then $u_{k-1} \ldots, u_1$. For the rectangles indicating deletion, we keep them the same. For the rectangles indication re-parenting, for each u_i we only consider the vertices to the right of u_{i-1}. We note that in the re-parenting step, any vertex returned will have parent u_i.

For each active vertex v satisfying deletion, we find the the first interval u_j such that $l_v \leq r_{u_j}$. As the effect of inserting u_l for $l \leq j$ is the same as only inserting u_j, we delete v from the range search, and add it back in when we are inserting u_j. If the current vertex we are inserting is u_j, then we apply the deletion criteria.

For each compressed path v found in re-parenting, since that would be deleted in the insertion of u_l with $l > i$ are removed from the range search data structure, we may assume that v is not deleted. Again by Lemma 6.6.13, we have $r_{u_{i-1}} < l_v \leq r_{u_i}$, and applying the effect on v from Lemma 6.6.13 now is will be the same as the effect on v from the insertion of all u_l for $l \geq i$. At this point, its parent is now u_i and we remove it from the range search data structure (otherwise it is returned from the range search data structure, but it is determined that its parent is also returned, at which point its action is that of being a descendant and it is deleted from the range search data structure).

By Lemma 6.6.13, each vertex v considered is either an active interval deleted - with parent changed or not, or is an interval that re-parents.

In the first case, when we find that a vertex v will be deleted, it found as a deleted vertex at most twice, since we delete it from the range search data structure, and add it back at I_{u_j} at which point it is either deleted for the last time (and its parent is decided) or it is re-parented.

For a vertex that re-parents, its new parent u_i for some i, and henceforth will never be considered since u_i will be and it is a descendant of u_i.

Therefore, any active node will be returned by the range search data structure at most 3 times - once when it is deleted for the first time, once if it is deleted from the insertion of its potential parent u_j and once when its compressed path needs to re-parent. Thus the time taken to change the parents of all active vertices is $O(|S| \lg |S|)$. □

Theorem 6.6.15. *Let G be an interval graph, and let $q_{t_1}, \ldots, q_{t_2}$ be a sequence of queries. Then there exists a reduced representation of G, $R(P(G, t_1, t_2), S(t_1, t_2))$ occupying $\tilde{O}(t_2 - t_1)$ words of space and supports* `batch_insert` *and* `reduce_active` *in time $O((t_2 - t_1) \lg(t_2 - t_1))$.*

Therefore, we have the main theorem of this section:

Theorem 6.6.16. *Let G be an interval graph, let $q_1, \ldots, q_t$ be a sequence of queries. Then we can output the result of all* `distance` *queries in $O(n \lg n + t \lg^2 t)$ time.*[5]

Proof. We construct $R(P(G, 1, t), S(1, t))$ by first finding the exposed vertices $\mathcal{I}^{\texttt{exposed}}(G)$, then inserting all the points in $S(1, t)$ into it. This take $O(n \lg n + t \lg t)$ time. Next compress the tree and construct the dynamic range reporting data structure in $O(n \lg n + t \lg t)$. We now apply Algorithm 1. Since `batch_insert` and `reduce_active` are both $O(|S| \lg |S|)$ time, the run time of Algorithm 1 is $O(t \lg^2 t)$ time.

Thus in total, the time cost is $O(n \lg n + t \lg^2 t)$ time. □

[5]We will assume that it $\lg t = O(\lg n)$ and if not, break the sequence of queries up into blocks of n queries.

6.7 Dynamic Interval Graph

In this section, we will consider the problem without restrictions on general interval graphs. We will break the interval graph up into smaller sub-interval graphs which will be easier to maintain. For any path, we may similarly break it into blocks of nodes belong to each of the smaller sub-interval graphs. We will compute the length of each of the sub-paths that falls in entirely in a sub-interval graph. By doing so, as the sub-interval graphs are of smaller size, we are able to maintain them at lower time cost.

Given an interval graph G by an interval representation $\mathcal{I}$ in sorted order, we are able to construct the distance tree (using a pointer representation of the tree in $O(n \lg n)$ bit of space) in $O(n)$ time. We can then construct a anc structure on it in $O(n)$ time as well [13], therefore we are able to answer all the necessarily queries on the distance tree (anc, depth) in $O(1)$ time after $O(n)$ preprocessing.

Let $S(n)$ (or just S for short) be a parameter. We break G into n/S blocks B of vertices by their left endpoints. So that block 1 (B_1) contains the first S vertices by left endpoint and so on. For each block B_i, the graph $G(B_i)$ is the induced subgraph of G on the vertices of B_i. For each block, we build a distance tree $T(B_i)$. We note that even if G were connected, each block $G(B_i)$ may not be. As we will now have many different distance trees, when we refer to parent, we refer to the parent of a vertex in G (that is in the distance tree $T(G)$). If we need to refer to the distance tree of a particular block, we will use $\texttt{parent}_{T(G(B_j))}$.

Definition 6.7.1. Let v be a vertex in G, we say that v has an ***in-block*** parent if v and $\texttt{parent}_T(v)$ are in the same block (or if v is the root of the distance tree of its component) and v has an ***out-of-block*** parent otherwise.

We note that by this definition, a vertex v belong to block B_i has an in-block parent implies that $\texttt{parent}(v) = \texttt{parent}_{T(G(B_i))}(v)$ (i.e, the vertices that these nodes represent are the same).

Example 6.7.2. Consider Figure 6.2. We see that all the vertices represented by the red intervals have in-block parents, while all other vertices do not. In particular, for any other vertex, since they have an out-of-block parent, their parent in the distance tree of their block $T(G(B_i))$ would be different than their parent over all. However for the red vertices these two parent relationships would coincide.

Let v be a vertex, and consider the the path from v to the root r of its component given by Lemma 4.4.2 as $v = v_1, \ldots, v_a = r$. We may break the path into a sequence of

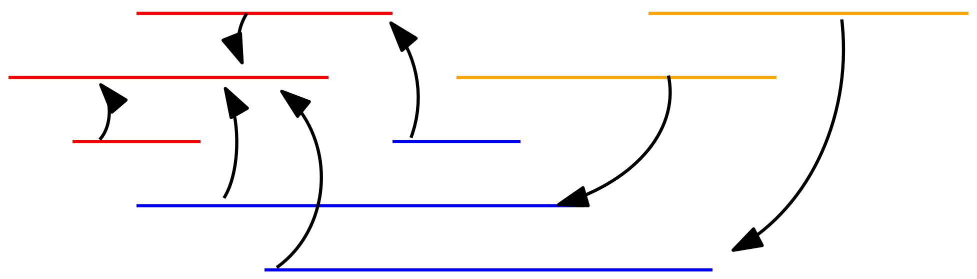


Figure 6.2: An interval graph with the parent relationship depicted by arrows, divided into blocks of size 3.

sub paths P_j where every vertex in P_j belongs to block B_j, which we will call the *block-compressed-path*, where every vertex $v_i \in P_j$ belong to block j. By this definition, every vertex $v_i \in P_j$ except the last have in-block parents, and the last vertex of every block has an out-of-block parent.

Lemma 6.7.3. *Let G be an interval graph, and v be a vertex. Let $P_{j_1}, \dots, P_{j_b}$ be the block-compressed-path of the path to the root (of the component of v) from v. Then for any index k, $j_k > j_{k+1}$.*

Proof. For any vertex v_i, we have $l_{v_i} > l_{v_{i+1}}$, so that v_{i+1} appears in the same or an earlier block. Thus $j_k > j_{k+1}$. □

The above lemma shows that the block numbers are monotonic decreasing. Thus since no block is used more than once, the number of blocks on any path is at most n/S. As in Lemma 4.4.2, given $u < v$ a shortest path is found by the first index i such that $l_{v_i} \leq r_u$. We will now consider how u interacts with the block-compressed-path.

In the first case, $u \in P_j$ for some j, and in the second, there is an index j_l such that $u \in B_k$ for some index k such that $j_{l+1} < k < j_l$ (here j_{l+1} may not exist if $j_l = j_b$ is the last block in the path). Now consider the first case:

Lemma 6.7.4. *Let $u, v \in B_j$ be two vertices in the same block. Let v_i be the first index on the path to the root with an out-of-block parent (that is $v_i > d(B_j)$ but $v_{i+1} \leq d(B_j)$). If $v_i \leq u$ then $\texttt{distance}_G(u, v) = \texttt{distance}_{B_j}(u, v)$. Otherwise, either u, v are not connected in G or v_{i+1} is adjacent to u.*

Proof. Consider the path to the root from v. Since $v_i \leq u$, then the first vertex on the path adjacent to u has smaller index, and thus is in B_j. By Lemma 4.4.2, the shortest path from v to u would lie entirely in B_j so $\texttt{distance}_G(u, v) = \texttt{distance}_{B_j}(u, v)$.

Now suppose that $v_i > u$. Suppose that u, v are connected in G. Then since v_i has an out-of-block parent, $v_{i+1} < u$. Thus $l_{v_{i+1}} < l_u$ and $r_{v_{i+1}} \geq l_{v_i} > l_u$. Thus u is adjacent to v_{i+1}. □

We next consider the second case:

Lemma 6.7.5. *Let $u < v$ be two vertices. Consider the block-compressed-path of v. Suppose that there exists an index j_l such that $u \in B_k$ and $j_{l+1} > k > j_l$. Let v_i be the last vertex in P_{j_l} on the block-compressed-path of v. Then u, v are connected if and only if l is not the last block on the path. If l is not the last block on the path then either v_i or v_{i+1} is adjacent to u and v_{i-1} is not adjacent to u.*

Proof. First suppose that j_l is the last block on the path. Then consider `parent`(v_i). If it belongs to block j_l, then v_i is not the last vertex of the path in that block. If it belongs to a different block then P_{j_l} is not the last block on the path. In either case, we have a contradiction. Thus v_i must be the root of the distance tree of its component, and $u < v_i$ implies that u is not adjacent to v_i and thus not adjacent to v. Conversely, we have that $l_{v_{i+1}} < l_u < l_{v_i} \leq r_{v_{i+1}}$ by the block numbers and that $v_{i+1} =$ `parent`(v_i). Hence v_{i+1} is adjacent to u. It is also possible that $r_u \geq l_{v_i}$, so that u is adjacent to v_i. Thus either v_i or v_{i+1} is adjacent to u.

Now consider v_{i-1}. If u is adjacent to v_{i-1}, then $r_u \geq l_{v_{i-1}}$, and thus `parent`$(v_{i-1}) \leq u < v_i$ contradicting the fact that $v_i =$ `parent`(v_{i-1}). □

Now suppose that given v, we are able to compute the last vertex on the path to the root belonging to the same block as v. Then we may iteratively compute this, then check whether it or its `parent` is adjacent to u and continue to the next block. We note that we may compute `parent` using the data structure in Section 6.3 for the vertices with out-of-block parents.

Thus we now consider how to compute the sub paths P_j. That is given a vertex v_i, we want to find the next vertex in the path $v_{i'}$ such that $v_{i'}$ has an out-of-block parent, which implies that P_j ends at $v_{i'}$. We will denote this operation as `out_of_block_ancestor` We first characterize the vertices of a block B_j whose parents are in-block and those vertices whose parents are out-of-block.

Lemma 6.7.6. *Let B_j be a block of vertices. Then there exists an index/vertex w such that every vertex $v \in B_j$ with $v > w$ have in-block parents and every vertex $v \leq w$ have out-of-block parents.*

Proof. We will show that if v has an in-block-parent, then so that $v+1$, and if vertex v has an out-of-block-parent, so does $v-1$.

Suppose that v has an in-block-parent $p_v = \texttt{parent}(v)$. Then $p_{v+1} = \texttt{parent}(v+1)$ is to the right of p_v by Lemma 4.4.1. Thus $p_v \leq p_{v+1} \leq v$ and p_{v+1} is in the same block as p_v, v and $v+1$. Suppose that p_v is out-of-block. Then as $p_{v-1} \leq p_v$, so must p_{v-1} and $v-1$ also has an out-of-block parent. □

We will denote this point/vertex as $d(B_j)$, where the vertices $v \leq d(B_j)$ have out-of-block parents and vertices $v > d(B_j)$ have in-block parents. To compute B_j we employ the following:

Lemma 6.7.7. *Let B_j be a block of vertices and let v be the first vertex of the block (that is the vertex with the smallest l_v). Then $w = d(B_j) = \arg\max\{l_w \mid l_w \leq r_u, w \in B_j\}$ where $u = \arg\max\{r_u \mid l_u < l_v\}$. We note that if u is such that $r_u < l_v$, then v would be the left most vertex of its connected component in G, and every vertex of B_j would have an in-block parent.*

Proof. Let $v' \in B_j$ with $v' \leq d(B_j)$. Then $l_{v'} \leq l_w \leq r_u$ and v' is adjacent to u. Since $\texttt{parent}(v') \leq u$, it must be in a different block and it has a out-of-block parent. Conversely suppose that v' has an out-of-block parent. Let $p = \texttt{parent}(v')$. We have $l_p < l_v$, and thus by our choice of u, $r_u \geq r_p$. Since $p = \texttt{parent}(v')$, we also have $l_{v'} \leq r_p \leq r_u$, hence $v' \leq w$ by our choice of w. □

Suppose that for a block B_j we have computed $d(B_j)$, then given a vertex $v \in B_j$, we can compute $v' = \texttt{out\_of\_block\_ancestor}(v)$ as follows:

Lemma 6.7.8. *Let $v \in B_j$ be a vertex with an in-block parent. Let $w = d(B_j)$ be the divider between vertices with in-block parents and those with out-of-block parents. Let v' the first ancestor of v with an out-of-block parent. Let $v = v_1, \ldots v_a$ be the path to the root in $T(B_j)$. By Lemma 4.4.2, there exists the first index i where $l_{v_i} \leq r_w$. Then $v' = v_i$ or $v' = v_{i+1}$.*

Proof. We note that $l_w \leq r_w < l_{v_{i-1}}$ so that $w < v_{i-1}$. Hence by Lemma 6.7.6 v_{i-1} has an in block parent. Now suppose that $l_{v_i} \leq l_w$. Then v_i has an out-of-block parent. Since v_{i-1} does not, then $v' = v_i$. On the other hand, suppose that $l_w < l_{v_i}$, then v_i has an in-block parent, but as $v_{i+1} \leq w$ (by definition of $\texttt{parent}_{T(G(B_j))}$), v_{i+1} has an out-of-block parent and $v' = v_{i+1}$. □

This lemma states that if we can compute $d(B_j)$, then we may compute `out_of_block_ancestor` using only `depth` and `anc`.

Our data structure is then as follows:

- We maintain a copy of the data structure in Section 6.3, as described in Theorem 6.3.1.
- We break the vertices of the graph into blocks B_j of size $O(S(n))$ which we will maintain, and a distance tree $T(G(B_j))$ on each block, storing a map between the vertices of the block and the nodes of the tree.

We will compute the distance as follows:

The correctness of Algorithm 2 follows from the previous lemmas and Lemma 4.4.2.

Now we show how to maintain the data structure under `insert` and `delete`.

Lemma 6.7.9. *Let G be divided into blocks B_j. Upon the insertion or deletion of an interval $w = [l_w, r_w]$, the blocks j where $d(B_j)$ may be updated are the blocks between $j_1, \ldots, j_2$, where $j_1 - 1$ is the block that w is inserted into (the block that w would be inserted into is determined by the ordering of the left endpoints, if it could be inserted in either of two adjacent blocks, we insert it as the first vertex of the larger indexed block) or deleted from, and j_2 is the block containing $\arg\max\{l_u \mid l_u \leq r_w\}$.*

Proof. By Lemma 6.7.7, for a block B_j, $d(B_j)$ is computed as $u = \arg\max\{r_u \mid l_u < l_v\}$ where v is the first vertex of the block. In order for w to be considered in the computation, we must have $l_w < l_v$, and $r_w \geq l_v$. Thus the last block to satisfy $r_w \geq l_v$, is the block containing $u = \arg\max\{l_u \mid l_u \leq r_w\}$. The first block considered is the first block whose first vertex v has $l_w < l_v$ which is the block following the block that w is inserted into. □

We note that by Lemma 6.7.7, upon an insertion aside from B_{j_2}, all other blocks would now only contain vertices with out-of-block parents.

Lemma 6.7.10. *We may support* `insert` *in $O(\lg n + S(n))$ time.*

Proof. By Theorem 6.3.1, inserting the interval into the navigational data structure takes $O(\lg n)$ time. We may also use it to determine the block we need to insert w into, and re-build the distance tree of that block in $O(S(n))$ time. If the size of the block exceeds $2S(n)$, we split it into two blocks and build the distance tree for both. □

```
input  : Dynamic Interval Graph $G$, two vertices $u < v$, where $u \in B_k, v \in B_{j_1}$
output: The distance between $u$ and $v$
$j \longleftarrow j_1$;
dist $\longleftarrow 0$;
while $True$ do
    $v' \longleftarrow$ OutOfBlockAncestor($v$);
    if $k = j$ then
        if $v' \leq u$ then
            return $dist$ +distance$_{G(B_j)}(u, v)$
        end
        if adjacent($u, v'$) then
            return $dist$ + distance$_{G(B_j)}(v, v') + 1$
        end
        if adjacent($u$, parent($v'$)) then
            return $dist$ + distance$_{G(B_j)}(v, v') + 2$
        end
        return Not connected
    end
    if adjacent($u, v'$) then
        return $dist$ + distance$_{G(B_j)}(v, v') + 1$
    end
    if $v' =$ parent($v'$) then
        return Not connected
    end
    if adjacent($u$, parent($v'$)) then
        return $dist$ + distance$_{G(B_j)}(v, v') + 2$
    end
    dist $\longleftarrow$ dist + distance$_{G(B_j)}(v, v') + 1$;
    $v = \longleftarrow$ parent($v'$);
    Set $j$ as the block of $v$;
end
```
Algorithm 2: Algorithm for computing the distance in dynamic interval graphs

Lemma 6.7.11. *We may support* `delete` *in* $O(\lg n + S(n))$ *time.*

Proof. By Theorem 6.3.1, deleting the interval from the navigational data structure takes $O(\lg n)$ time. We may also use it to determine the block that w belongs to, and re-build the distance tree of that block in $O(S(n))$ time. If the block drops below $S(n)/2$, then we merge it with one of the adjacent blocks (and split it if the new block exceeds $2S(n)$ vertices). We then rebuild the appropriate distance trees for the new blocks. □

Lemma 6.7.12. *We may support* `distance` *in* $O(n \lg n/S(n))$ *time.*

Proof. Consider Algorithm 2. The loop runs at most $n/S(n)$ times since the number of blocks in the block-compressed-path is at most $n/S(n)$. The time required per loop is dominated by the computation of `out_of_block_ancestor`. By Lemma 6.7.8, the time cost is $O(1)$ plus the time to compute $d(B_j)$ for the block B_j. The formula can be computed by searching for l_v where v is the first vertex in B_j in the tree of left end points in Theorem 6.3.1, then among the smaller vertices finding the one with maximal right endpoint. This takes $O(\lg n)$ time. □

Lastly, we handle the case when we insert or delete many vertices, so that the number of blocks and the size of the blocks may no longer be $O(S(n))$. When this occurs, we rebuild blocks and the distance trees. Let n' be the size of the graph at the time of the previous rebuild, and n be the current size of the graph. If $n = 2n'$ or $n = n'/2$ we rebuild the distance trees using $O(n)$ time. Thus this would require $O(1)$ amortized cost to both `insert` and `delete`.

Theorem 6.7.13. *Let G be an interval graph and let $S(n)$ be a parameter. There is a data structure for G occupying $O(n \lg n)$ bit of space which supports* `adjacent`, `degree` *in* $O(\lg n)$ *time,* `neighborhood`, `spath` *in* $O(\lg n)$ *time per vertex returned,* `insert`, `delete` *in* $O(\lg n + S(n))$ *time and* `distance` *in* $O(n \lg n/S(n))$ *time.*

In particular, if we set $S(n) = \sqrt{n \lg n}$, *then* `insert`, `delete`, `distance` *all have time complexity* $O(\sqrt{n \lg n})$.

Proof. We have shown the above with amortized time bounds. To de-amortize at the time of a rebuild, we keep the current data structure as is. We begin the construction of the new data structure at the new block size $S(n)$. Over the next $n'/4$ updates, we construct the new data structure using time $O(n)$ and perform the $n'/4$ updates, using time at most $O(nS(n))$ time. At then end of the $n'/4$ updates, we switch to answering queries on the newly constructed data structure. In the mean time, we perform all queries and updates on the old data structure. On each of the next $n'/4$ updates, we perform $O(1/n)$ of the work to build the new data structure, so that at the end, we will have the new data structure completed. Therefore on each update, we only need to perform $O(S(n))$ extra work, and the time complexity is worst case. □

6.8 Discussion

In this chapter we considered many variations of the dynamic interval graph problem. Due the graph being dynamic and the fact that we do not choose the end points of intervals, our space cost usage is $O(n \lg n)$ bits, even in the proper interval graph case.

We showed that due to the operations being local (we only need to search for adjacent vertices), we are able to support `adjacent`, `degree`, `neighborhood`, `spath` in $O(\lg n)$ time (per node or value outputted) under both `insert` and `delete`, and it can be done by simply storing balanced binary search trees.

We next considered supporting `distance` query. First we studied the case of proper interval graphs. By applying the ordinal-binary tree isomorphism, we were able to reduce the number of edges changed upon an update to $O(1)$ so that updates can be supported in $O(\lg n)$ time while supporting `distance` in $O(\lg n)$ time as well.

We then considered the case that we only need to support `insert` or `delete` but not both. We showed that in an interval graph, it suffices to only maintain a proper interval graph of the exposed vertices. However any particular `insert` or `delete` could change this proper interval graph by a large amount. However, over many updates, the total amount of change is bounded, and thus the amortized cost of `insert` and `delete` can be bounded by $O(\lg n)$.

Our next problem tackled the offline version of the problem, where all the updates and queries are given in advance. To solve this, we employed divide and conquer on time, to reduce the problem to a large static graph with a small number of updates. We then reduced our representation of this large static graph so that it's size is proportional to just the number of updates, which allowed us to make the updates quickly. For t queries, we were able to compute the answers in $t \lg^2 t$ time.

Finally we dropped all the restrictions. We broke the graph into subgraphs of smaller size which are easily maintained, and computed the path within each subgraph. In doing so, we were able to support `insert`, `delete` and `distance` in $O(\sqrt{n \lg n})$ time.

Our results in this chapter are probably not optimal, especially in the case of fully dynamic interval graphs. The time complexity of $O(\sqrt{n \lg n})$ is much higher than the time for all the other versions of the problem, which were $\tilde{O}(n)$. Thus further work on improving the the data structure in this case is needed. For smaller improvements, the data structure for incremental and decremental interval graphs is amortized $O(\lg n)$, since any particular update may change the graph in a large way. It would be desirable to de-amortize it, so that it is $O(\lg n)$ worst case time.

Chapter 7

Chordal Graphs and Path Graphs

In this chapter we will study data structure problems for path graphs and chordal graphs. In the previous chapters, we studied various data structure problems on interval graphs. Path graphs and chordal graphs are in a sense, generalizations of interval graphs. Whereas interval graphs are intersection graphs of paths in a path, path graphs generalize this to paths in a tree, and chordal graphs generalize this further to subtrees in a tree. In this chapter, we will study the navigational queries and distances in the static case. The chapter is organized as follows: we begin with a brief review of relevant previous works in Section 7.1, and an overview of existing results that we will be using in Section 7.2. We will study study chordal graphs in Section 7.3 and path graphs in Section 7.4. Finally, we will summarize our results and discuss open problems in Section 7.5.

7.1 Previous Work

On path graphs, they were studied by Gavril [36], who gave a recognition algorithm for them. The data structure problem was studied by Balakrishnan et al. [8], where gave gave both succinct and compact data structures. Specifically, they gave a succinct $n \lg n + o(n \lg n)$ bit data structure supporting `adjacent` in $O(\lg^2 n)$ time, and `degree` and `neighborhood` in $O(\lg^2 n)$ time per neighbour. To complete the time-space trade off, they gave a data structure using $O(n \lg^2 n)$ bits supporting `adjacent` in $O(1)$ time, `degree` and `neighborhood` in $O(1)$ time per neighbour.

There have been much more studies of chordal graphs, as they naturally arise in many situations. For example, in the study of Gaussian Elimination of matrices [73], which gives rise to the term *perfect elimination order* [1]. In the studied of data structures, Singh et al. [76] gave a $O(n \lg n)$ bit data structure to compute `distance` approximately in $O(1)$ time. If d is the distance between two vertices u, v, then their data structure returned an answer in the range $[d, 2d+8]$. Munro and Wu [65] [2] gave a succinct data structure occupying $n^2/4 + o(n^2)$ bits to support `adjacent` in $O(f(n))$ time, `degree` in $O(1)$ time, `neighborhood` in $O((f(n))^2)$ time, and `distance` and `spath` in $O(nf(n))$ time for any $f = \omega(1)$. Furthermore, Munro and Wu constructed a data structure for approximate distances occupying $O(n \lg n)$ bits which supported reporting the approximate `distance` in $O(1)$ time. They range is also reduced from $[d, 2d+8]$ to $[d, d+1]$.

7.2 Preliminaries

As always, one of the fundamental building blocks of succinct data structures is the bit vector.

Lemma 2.3.2 ([62])**.** *A bit-vector of length n can be represented in $n+o(n)$ bits to support* `access`, `rank`, `select` *in $O(1)$ time.*

As the graphs are defined by the intersections of paths and subtrees in a tree, naturally we will require succinct tree data structures. In this case, we do not need the level-order operations implemented in Chapter 3.

Theorem 3.5.2 (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 2.1 in $O(1)$ time.*

We will also rely heavily on the orthogonal range search data structures.

Lemma 2.5.6. *Given n 2-dimensional points, and let f be the time cost of* `rank`*, g be the time cost of* `decode`*. Then we are able to support 3-sided reporting queries (of the form $[x_1, x_2] \times [-\infty, y]$, to support the symmetric cases, we duplicate the structure) in time $O(f + k \cdot g)$ where k is the number of points reported. The space cost is $2n + o(n)$ plus the space needed to support the* `rank` *and* `decode` *operations.*

Lemma 2.5.3 ([16])**.** *Let S be a set of points from the universe $M = [1..n] \times [1..n]$, where $n = |S|$. S can be represented using $n \lg n + o(n \lg n)$ bits to support orthogonal range counting in $O(\lg n / \lg \lg n)$ time, and orthogonal range reporting in $O(k \lg n / \lg \lg n)$ time, where k is the size of the output.*

[1]An ordering of vertices such that for every vertex, its adjacent vertices that before it is a clique.
[2]This work appears in the author's M.Math thesis.

And finally permutations.

Lemma 2.3.8 ([60]). *Let P be a permutation. Then we may represent P using $(1 + 1/f(n))n \lg n + o(n \lg n)$ bits to support the computation of P and P^{-1} in $O(f(n))$ time ($1 \leq f(n) \leq n$). In particular, if we set $1/f(n) = \varepsilon$ for constant $\varepsilon > 0$, then the space is $(1 + \varepsilon)n \lg n + o(n \lg n)$ bits and the time is $O(\frac{1}{\varepsilon}) = O(1)$.*

7.3 Chordal Graph

A chordal graph is [3] the intersection graph of subtrees of a tree. For a chordal graph G, we may find a tree T such that for every vertex v, we associate v with a subtree X_v such that two vertices u, v are adjacent if and only if X_v intersects X_u at a node.

In this section, we revisit the data structure problem studied by Munro and Wu [65]. In particular, we give a succinct data structure occupying $n^2/4 + o(n^2)$ bit which supports `adjacent` in $O(1)$ time (rather than $O(f(n)))$ time, `degree` in $O(1)$ time, `neighborhood` in $O(f(n))$ time (rather than $O((f(n))^2)))$, and `distance` and `spath` in $O(n)$ time (rather than $O(nf(n))))$.

To start, we will briefly review the result of [65]. First we assume that the graph is connected, and if not apply the following to each connected component. As it is shown, the tree T can be chosen to have the following properties:

- T has n nodes
- For every vertex v, the node of X_v denoted by T_v with the smallest depth is unique to X_v. That is for any two vertices u, v, T_u and T_v are distinct.
- For every vertex v, let u be the vertex such that $T_u = \texttt{parent}(T_v)$. Then u is adjacent to v (that is T_v is in the subtree X_u)

The first two properties allows us to associate every vertex v with a node T_v in the tree, and we will now abuse notation and call both v and T_v v. We may also now name each vertex with a number, which corresponds to its index in the preorder traversal of T. Munro and Wu also defined [4] the set at v, $B(v)$ to be the set of subtrees (not including v) passing through v. By property 3, $\texttt{parent}(v) \in B(v)$.

[3]Using one of many characterizations of chordal graphs

[4]Not exactly, but as an immediate consequence.

Furthermore, as a consequence of this structure, we have that $B(v) \subseteq B(\texttt{parent}(v)) \cup \{\texttt{parent}(v)\}$ since any path passing through v must also pass through $\texttt{parent}(v)$. Therefore the elements of $B(v)$ are a subset of the ancestors of v, and thus the elements of $B(v)$ are smaller (in the sense of the index in preorder traversal) than v.

Munro and Wu defined the following operations for $B(v)$:

- `adjacent`, given $u < v$, is $u \in B(v)$? We note that this is exactly `adjacent` since $u \in B(v)$ if and only if u is adjacent to v.
- `decode`, given v and an index i, what is the i-th smallest element of $B(v)$?

Using these, Munro and Wu constructed the data structure for chordal graphs as follows:

- `adjacent`: This is clear.
- `degree`: We store the degree of every vertex explicitly in $n \lg n$ bit of space.
- `neighborhood`: The neighbours u of v with $u < v$ can be found by iterating `decode`. The neighbours u of v with $u > v$ can be found by compressing the following bitvector. We consider a bitvector of length $n - v$ where the i-th bit is a 1 if the vertex $v + i$ is a neighbour of v. We compress the bitvector into a smaller bitvector of length $(n - v)/f(n)$ by considering blocks of size $f(n)$. The value of the block is a 1 if there is a 1 in the block. To find all the 1s in the bitvector, we iterate over the 1s in the compressed bitvector, and for each 1, check every vertex in that block if it is adjacent to v. Thus we incur a factor of $f(n)$ (since we need to scan the block) for every vertex reported in this case.
- **`spath` and `distance`**: We use a generalized lemma as Lemma 4.4.2. We again construct the distance tree T_D, with the parent relationship $\texttt{parent}_{T_D}(v) = \min\{B(v)\}$, which is the minimal adjacent ancestor of v. We assume that u, v are not in an ancestor-descendant relationship, as in the case they they are in an ancestor-descendant relationship, the graphs restricted to the path between them is an interval graph, which is covered in Lemma 4.4.2.

Lemma 7.3.1. *Let $u < v$ be two vertices of a chordal graph G, with tree T. Let $l = \texttt{LCA}_T(u, v)$, and assume that $l \neq u$. Consider the two paths to the root $u = u_{k_1}, \dots u_0 = r$ and $v = v_{k_2}, \dots, v_0 = r$. Let i, j be the indices where $u_i > l \geq u_{i-1}$, $v_j > l \geq v_{j-1}$. Then there are two cases:*

1. *If there exists a vertex w such that w is adjacent to both u_i and v_j (i.e. the subtree X_w contains both the nodes T_{u_i} and T_{v_j}), then a path is $u = u_{k_1}, \ldots, u_i, w, v_j, \ldots, v_{k_2} = v$.*

2. *If no such vertex w exists, then a shortest path is $u = u_{k_1} \ldots, u_i, u_{i-1}, v_{j-1}, \ldots v_{k_2} = v$.*

We store T_D and a mapping between the vertices v in T and the corresponding node of T_D. To compute a shortest path or a distance, we compute the `LCA` of u, v, and find u_i and v_j. As in the interval graph case, u_i and v_j can be found using level-ancestor. If w exists, then $w \in B(u_i) \cap B(v_j)$, and we find it by computing this intersection in $O(n)$ applications of `decode` - this also determines if w does not exist.

Putting this together, we have the following theorem of Munro and Wu:

Theorem 7.3.2. *Let G be a chordal graph. Let D be a data structure answering* `adjacent`, `decode` *in $O(g(n))$ time, which occupies $n^2/4 + o(n^2)$ bits of space. Then we may support* `adjacent` *in $O(g(n))$ time,* `degree` *in $O(1)$ time,* `neighborhood` *in $O(f(n)g(n))$ time for any function $f(n) \in \omega(1)$, and* `spath`, `distance` *in $O(ng(n))$ time. The space is succinct: $n^2/4 + o(n^2)$ bits.*

Furthermore, we may support approximate shortest paths and distances (which gives an answer in the range $[d, d+1]$) using $O(n \lg n)$ bits of space and $O(1)$ time.

Proof. The lower bound in space is $n^2/4$ bits, so D would need $n^2/4$ bits in the worst case. The additional space required for `neighborhood` are bitvectors whose total length is $O(n^2/f(n)) = o(n^2)$.

To compute approximate shortest paths and distances, we only need to store T, T_D and the mapping between them, as we do need to need to determine whether w exists in Lemma 7.3.1, which takes $O(n \lg n)$ bits of space. □

Munro and Wu showed how to construct D with $g(n) = \omega(1)$ and set $f = g$. We will show how to construct D with $g(n) = O(1)$.

We select $O(\sqrt{n})$ nodes of the tree so that every node has at least one such node among its $\sqrt{n}$ immediate ancestors. One way to do this is applying the same Lemma 5 used in [65] to compute a k-path vertex cover. A second way to do this follows from subsection 3.4.1 by selecting the offset i that minimizes the nodes at levels l with $l = i \mod \sqrt{n}$. Following [65], we will denote these nodes as *shortcut* nodes.

For these shortcut nodes, we store the sets $B(v)$ explicitly using n bits of space as a bitvector, so that we can answer the set membership queries using `rank`, `select` and `access`.

For a node v with parent u, we say v is enlarging if $B(v) = B(u) \cup \{u\}$. We note that as $B(v) \subseteq B(u) \cup \{u\}$, $|B(v)| \leq |B(u)| + 1$ with equality exactly when v is enlarging.

For an enlarging vertex v, we find the ancestor u of v of greatest depth that is not enlarging - so that all nodes between u and v are enlarging. If there is a shortcut node between u and v, we take the shortcut node instead. We store a pointer to that node. Let $u_1, \ldots, u_k$ be the nodes between u and v. Then by definition, $k \leq \sqrt{n}$ and $B(v) \subset B(u) \cup \{u, u_1, \ldots, u_k\}$. Since each u_i are enlarging, we actually have $B(v) = B(u) \cup \{u, u_1, \ldots, u_k\}$, and thus we do not need to store anything as long as we can answer the queries on $B(u)$.

For a non-enlarging vertex v, let u be the greatest depth shortcut ancestor. Again let $u_1, \ldots, u_k$ be the nodes between u and v. By definition, $k \leq \sqrt{n}$ and $B(v) \subset B(u) \cup \{u, u_1, \ldots, u_k\}$. We store this as a length $|B(u)| + k \leq |B(u)| + \sqrt{n}$ bit vector.

Furthermore, we store the size of each of the sets $B(v)$, using $\lg n$ bits at each node. The total space for sizes is $O(n \lg n)$ bits.

The space used is $O(n\sqrt{n})$ bits for the shortcut nodes.

Let M be the maximum size of $|B(v)|$ for a shortcut node v, then we must have at least M enlarging vertices and at most $n - M$ non-enlarging vertices. The enlarging vertices store just a pointer, so we use at most $n \lg n$ bits of space. The non-enlarging vertices use at most $M + \sqrt{n}$ bits of space each, totaling at most $(1 + o(1))(n - M)(M + \sqrt{n}) \leq n\sqrt{n} + (1 + o(1))(n - M)(M) \leq n\sqrt{n} + (1 + o(1))n^2/4$ bits of space, maximized when $M = n/2$.

Now we show how to answer the two queries: `adjacent`, `decode`.

For `adjacent`(x, v) we break it up into two cases: v is enlarging or v is not enlarging. First we assume that v is not a shortcut node, as we stored the set explicitly and we can answer the query using access on the bitvector.

If v is not enlarging, let u be the nearest shortcut ancestor of v. We have $B(v) \subset B(u) \cup \{u, u_1, \ldots, u_k\}$, where u_i are the nodes between u and v. By checking the depths of nodes, we can see if $x \in \{u, u_1, \ldots, u_k\}$ and if so, check the corresponding bit in the bit vector. If not, we check if $x \in B(u)$, and if so, find the rank of x (i.e. x is the i-th smallest node in $B(u)$). We then check the i-th bit in the bitvector at v as this bit corresponds to the i-th smallest element in $B(u)$.

If v is enlarging, use the pointer to get the non-enlarging node u. We have $B(v) = B(u) \cup \{u, u_1, \ldots, u_k\}$, where u_i are the nodes between v and u on the tree. Again by a depth argument, we check if $x \in \{u, u_1, \ldots, u_k\}$. Otherwise we check if $x \in B(u)$. By construction, u is either a shortcut node or is non-enlarging, and we use one of the two previous cases.

For `decode`(v, i) we again break it into two cases v is enlarging or not. As above, if v is a shortcut node, we find the i-th smallest element of $B(v)$ by using select on the bitvector.

If v is not enlarging, let u be the nearest shortcut ancestor of v. We have $B(v) \subset B(u) \cup \{u, u_1, \ldots, u_k\}$, where u_i are the nodes between u and v. By selecting the i-th 1 in the bitvector of v, we can decide if it lies in the range $\{u, u_1, \ldots, u_k\}$. If so, we return the appropriate vertex. If it does not, then it corresponds to a node in $B(u)$. If j is the index of the i-th 1, then we are looking for the j-th smallest element of $B(u)$. Since u is a shortcut node, this is `decode`(u, j) and is handled above.

If v is enlarging, use the pointer to get the non-enlarging node u. We have $B(v) = B(u) \cup \{u, u_1, \ldots, u_k\}$, where u_i are the nodes between v and u on the tree. By checking the sizes of the sets, we decide if it lies in the range $\{u, u_1, \ldots, u_k\}$, and if so, return the appropriate vertex. Otherwise, it corresponds to a node in $B(u)$, and we need to return the i-th smallest element of $B(u)$. By construction, u is either a shortcut node or is non-enlarging, and we use one of the two previous cases.

As all the operations are simply bitvector operations, the run time of each are $O(1)$. Finally we apply Theorem 7.3.2.

Theorem 7.3.3. *Let G be a chordal graph. There is a succinct data structure occupying $n^2/4 + o(n^2)$ bits of space which supports* `adjacent` *in $O(1)$ time,* `degree` *in $O(1)$ time,* `neighborhood` *in $O(g(n))$ time for any function $g(n) \in \omega(1)$, and* `spath`, `distance` *in $O(n)$ time.*

7.4 Path Graph

In this section, we will give two data structures for supporting `adjacent`, `degree`, `neighborhood`. The lower bound for path graphs is $n \lg n - o(n \lg n)$ bits, inherited from the subclass of interval graphs. A matching upper bound is given by [8]. First we will give a succinct data structure with $O(\lg n / \lg \lg n)$ query times. We will follow it with a compact data structure using $(2 + \varepsilon)n \lg n$ bits but with $O(1)$ query times.

Tree Structure: Given a path graph G, we may find a intersection model for G using a tree T and a set of paths of T. As Gavril [36] showed, this tree can be chosen so that it has at most n nodes (corresponding to the maximal cliques of the graph G). Each vertex v is associated with a path P_v in T, and by definition, two vertices u, v are adjacent in G if the associated paths P_v, P_u intersect at some node of T. We will make several modifications to T while preserving the path intersections. This will allow our data structures to be simpler as we do not need to worry about multiple paths with the same endpoint.

First we root the tree arbitrarily. Next consider any endpoint of any path P_v at a node w of T. Create a child of w and extend the path to the child. We note that since P_v is the only path through the child, no additional intersections are created. By doing so, we add $2n$ nodes to T and guarantee that all endpoints of paths are unique.

Next for each internal node w, add a child node c_w as its last child. Consider the preorder numbers of the nodes in the subtree rooted at w. The smallest number is w and the largest is c_w. This also has the nice property that, if we are given a node c and are told that it is the largest preorder numbered node of some subtree (that is not the subtree rooted at c in the case that c is a leaf), we know that this subtree must be the one rooted at the parent of c. This at most adds another n nodes to T.

Therefore, after these modifications, our tree T has at most $4n$ nodes, with the property that

- For any two vertices u, v, the end points of the path associated with u, v, P_u, P_v are distinct nodes in the tree.
- For any internal node in the tree w, The last node in preorder in its subtree is the last child of w, and that child is necessarily a leaf. Conversely, if a leaf in the tree w is the last node in preorder of a subtree rooted at some vertex $x \neq w$, then x is the parent of w.

We will refer to the nodes of T by their preorder numbers. Each vertex v is associated with a path $P_v = (l_v, r_v)$ in T, where $l_v < r_v$ are the two end points of the paths. We will sort the paths based on the values of the left endpoints, and say that vertex $v \in [1, n]$ has the v-th smallest (as end points are unique, we do not have ties) left endpoint. We will abuse notation and use v to refer to the vertex in G, and the path P_v as well.

For a path $v = (l_v, r_v)$, denote the apex a_v as the node $\texttt{LCA}(l_v, r_v)$ in T. We note that since we extended the paths to "dummy" vertices, $a_v \neq l_v, r_v$. Given the apex of the path, denote the last node (by preorder numbers, and by construction this is the last child) in the subtree rooted at a_v by z_v. We will also use the function notation $a(v) = a_v$.

We will first describe how to compute the queries in the abstract in the next subsection, and then discuss the data structures and the concrete queries in the following two subsections.

7.4.1 Method

The main problem here is deciding on which criteria to use to decide whether two paths intersect. We note that in [8], they decomposed the tree using a heavy-light decomposition, and with it all the paths corresponding to the vertices. To take advantage of this, as they are now dealing with paths, the graphs restricted to them are interval graphs, and they can leverage the previous results (i.e. the data structure of Acan et al. [2]) on them. They then give criteria for each piece of the path based on whether they are on heavy paths or light edges. We will give a simpler global criteria on how to compute whether two paths intersect, and in doing so, reduce the amount of space needed and the time required. The criteria we will use is exactly what will be used in the `adjacent` query: given two vertices u, v, return whether they are adjacent or not. To do this, we consider a few cases based on the relationship between a_u, a_v.

- Suppose that a_u, a_v have no ancestor/descendant relationship, then as the paths u, v are contained within the subtrees rooted at a_u and a_v respectively, they cannot intersect and u, v are not adjacent.
- Suppose that $a_u = a_v$, then clearly u is adjacent to v.
- Suppose that a_u is an ancestor of a_v, u is adjacent to v iff exactly one of l_u, r_u is a descendant of (or equivalently, is in the subtree rooted at) a_v. To see this, note that since l_u, r_u must belong to subtrees rooted at different children of a_u, we cannot have both be descendants of a_v (as that would require them to belong to the same branch). One direction is clear as if one is a descendant of a_v then the paths would intersect at a_v. Conversely, wlog, suppose that P_v intersects the subpath l_u, a_u. Then some node $w \in P_v$ belongs to both paths, and thus l_u is a descendant of w which itself is a descendant of a_v.

We note that all the descendants of a node a_u have preorder numbers between a_u and z_u, and thus the ancestor/descendant relationships can easily be checked by comparing the preorder numbers of nodes.

Next we consider the `neighborhood` query. Given a vertex v, we wish to output all vertices u that are adjacent to v. We will use the `adjacent` criterion to filter the vertices using range reporting queries.

Case 1: $a_u = a_v$: we need to support the following operation: given an apex a, list out all paths u with it as its apex. We will denote this operation as `apex_list`(a).

Case 2: a_u is an ancestor of a_v: we need to support the following criteria - list out all paths u such that exactly one of l_u, r_u is a descendant of a_v. This can be expressed as the rectangles $(-\infty, a_v) \times (a_v, z_v)$ and $(a_v, z_v) \times (z_v, \infty)$.

Case 3: a_u is a descendant of a_v: we need to support the following criteria - list out all path u such that exactly one of l_v, r_v is a descendant of a_u. If we have the points (a_u, z_u), then the paths u are those with points in the rectangles: $(-\infty, l_v) \times (l_v, r_v)$ and $(l_v, r_v) \times (r_v, \infty)$

We note that the criteria from cases 2 and 3 are easily expressible in terms of 3-sided rectangles in a 2-dimensional range query. Thus it remains to realize these data structures.

7.4.2 Succinct Data Structure

The first data structure will use 4 sided queries mainly to implement `apex_list`. Because of this, the time complexity will be $O(\lg n / \lg\lg n)$.

First we further modify the tree in the following way: for each internal node a, consider all paths with a as its apex. Each path passes through two children of a, and in particular mark the child of a that the left branch of the path passes through. Reorder the children of a such that marked children come before unmarked children (in a stable manner, the order of any two marked children should not be changed - thus for any path, the left branch remains to the left of the right branch).

We will store a bitvector L of length at most $4n$ such that $L[i] = 1$ if the i-th node in preorder is the left end point of some path and 0 otherwise. We will store a bitvector R of length at most $4n$ such that $R[i] = 1$ if the i-th node in preorder is the right end point of some path and 0 otherwise. Finally, we store bitvectors A, Z such that $A[i] = 1$ if the i-th node is the apex of some path, and $Z[i] = 1$ if the i-th node is z_v for some path v, and 0 otherwise. Since we do not know which left end point matches with which right endpoint, we will consider the permutation $P[i] = j$ where the i-th path is $l_i = \texttt{select}(L, i), r_i = \texttt{select}(R, j)$. This encodes the statement that right endpoint of the i-th path is the jth largest right end point. The total space for these bit vectors is at most $16n$ bits.

We note that given a vertex v, we may compute the pertinent features of the path by: $l_v = \texttt{select}(L, v), r_v = \texttt{select}(R, P[v]), a_v = \texttt{LCA}(l_v, r_v)$ and $z_v = \texttt{last\_child}(a_v)$. All of these operations are $O(1)$ time except $P[v]$.

We will store this permutation in a 2-dimensional 4-sided range search data structure RS, with the points $(i, P[i])$ using $n \lg n + o(n \lg n)$ bits. Given a rectangle using preorder numbers as its coordinates, we will need to convert it into rank space that is stored. For example, the range $[x_1, x_2]$ will need to be converted to $[r_1, r_2]$ where $r_1 = \texttt{rank}(L, x_1 - 1) + 1$ (rank of the first left endpoint in the range) and $r_2 = \texttt{rank}(L, x_2)$ (rank of the last left endpoint in the range). Similarly for the y-coordinate using the bitvector R. We note that to compute $P[v]$, we need to report the single point in the rectangle $[v, v] \times [-\infty, \infty]$ using $O(\lg n / \lg\lg n)$ time. We will store the tree T succinctly, using $8n + o(n)$ bits.

Example 7.4.1. Consider the graph in Figure 7.1, which has 4 vertices on a tree with 19 nodes (which as mentioned we did not start with the optimal tree to show the paths, as many paths on the optimal tree are trivial paths, which begin and end at the same node). The relevant numbers for each paths are as follows:

Path v	l_v	r_v	a_v	z_v
Red Path	5	17	1	19
Blue Path	3	14	1	19
Green Path	13	18	1	19
Orange Path	9	11	7	15

The bitvectors would be:

- $L = 0010100010001000000$ $R = 0000000000100100110$
- $A = 1000001000000000000$ $Z = 0000010000000010001$

The permutation is $P[1] = 2, P[2] = 3, P[3] = 1, P[4] = 4$. The blue path's left endpoint is node 3, so it has the smallest left endpoint. The blue path's right endpoint is 14, which is the second smallest right endpoint, so $P[1] = 2$.

As an example of `adjacent`, to check if the green path and orange paths are adjacent, we first check their apex: 1 and 7. Since the green path's apex is higher, we check that exactly one of the endpoints is in the subtree rooted at node 7. These nodes have pre-order numbers between 7 and 15. Since the left endpoint numbered 13 satisfies this, they are adjacent.

The `apex_list` operation would return the red, blue and green paths (or rather the preorder numbers of the left endpoint, 5, 3, 13 as we identify the paths by their left endpoints) for the apex 1 and the orange path for the apex 7. For the apex 1, we check the following rectangles: $[2, 6] \times [7, 19]$, then $[7, 15] \times [16, 19]$ then $[16, 17] \times [18, 19]$. The first rectangle gives us the blue path (3,14) and red path (5,17), the second gives the green path (13,18), and the third is empty, so we stop there.

Implementing `adjacent` Query: Given vertices u, v, we obtain l_u, r_u, l_v, r_v, a_u, z_u, a_v, z_v as above, taking $O(\lg n / \lg\lg n)$ time. We then check the adjacency criteria in $O(1)$ time. Thus adjacency can be supported in $O(\lg n / \lg\lg n)$ time.

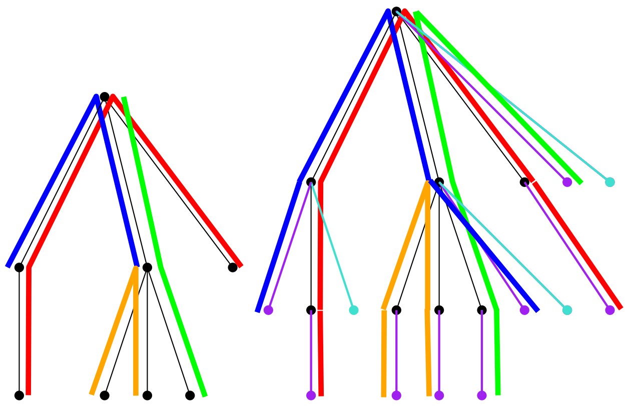


Figure 7.1: A path graph with 4 vertices. Note that for clarity, this is not the minimal tree that can be used for representing the graph (the minimal tree would consist of two nodes v_1, v_2 connected by an edge. The red path would only contain v_1, the orange only contain v_2 and blue and green would contain both). The four vertices' paths are coloured blue, red, green and orange. On the right is our modification to the tree. All endpoints are extended to the new purple nodes, all internal nodes have a new child added which is depicted by the teal node.

Implementing `neighborhood` Query: We consider the three cases from subsection 7.4.1. First, we will need to implement `apex_list`.

Let $w_1, \ldots, w_k$ be the children of a and let $z_1, \ldots, z_k$ denote the last node in the subtree rooted at $w_1, \ldots, w_k$. A path with apex a has its two branches in different children of a. Thus we will capture this by the rectangles $[w_i, z_i] \times [w_{i+1}, z]$ in the range search data structure RS, which captures the fact that the left endpoint of the path is a descendant of w_i and the right endpoint is a descent of one of $w_{i+1}, \ldots, w_k$. We stop once one of the rectangles is empty. By construction, the non-empty rectangles will be at the start. This allows us to get each path in $O(\lg n/\lg\lg n)$ time.

Case 1: $a_u = a_v$: we need to support the following operation: given an apex a, list out all paths u with it as its apex. We will denote this operation as `apex_list`(a).

Case 2: a_u is an ancestor of a_v: we need to support the following criteria - list out all paths u such that exactly one of l_u, r_u is a descendant of a_v. Suppose that we have n points generated from the paths as (l_u, r_u), then the paths satisfying the criteria are exactly those in one of these rectangles: $(-\infty, a_v) \times (a_v, z_v)$ and $(a_v, z_v) \times (z_v, \infty)$. Any point in the first rectangle satisfy the inequalities $-\infty < l_u < a_v < r_u < z_v$, which states that only r_u is a descendant of a_v, similarly for the second rectangle.

Case 3: a_u is a descendant of a_v: we need to support the following criteria - list out all path u such that exactly one of l_v, r_v is a descendant of a_u. Suppose that we have the at most n points generated from the paths as (a_u, z_u) (removing duplicates which is why it is at most n points), then the paths satisfying the criteria are exactly those with apex in one of these rectangles: $(-\infty, l_v) \times (l_v, r_v)$ and $(l_v, r_v) \times (r_v, \infty)$. Any point in the first rectangle satisfies the inequalities $-\infty < a_u < l_v < z_u < r_v$, which states that only l_v is a descendant of a_u, similarly for the second rectangle.

Thus we obtain the following theorem:

Theorem 7.4.2. *Let G be a path graph. We may represent G using $n \lg n + o(n \lg n)$ bits to support* `adjacent` *in $O(\lg n / \lg\lg n)$ time and* `neighborhood` *using $O(\lg n / \lg\lg n)$ time per neighbour. We may also support* `degree` *in $O(d \lg n / \lg\lg n)$ time.*

Proof. The only thing we did not discuss above is `degree`. However, finding the degree of a vertex follows from `neighborhood`, by finding all of its neighbhours and counting. The total space required is:

- The tree T, the bitvectors L, R, A, Z, the 2-dimensional 3-sided range reporting data structures, stored succinctly: $O(n)$ bits [5].
- The 2-D 4 sided range reporting data structure RS: $n \lg n + o(n \lg n)$ bits.

□

7.4.3 Compact Data Structure

In this subsection, we will improve the runtime of the previous data structure at the cost of extra space. We note that the time complexity arises because we stored the permutation P in a range search data structure because we needed 4-sided queries for `apex_list`. If we are able to implement `apex_list` in a faster way, we can store the permutation so that it can be computed more efficiently.

We keep the bit vectors L, R, A, Z as before. We store the permutation P using $(1 + 1/f(n))n \lg n + o(n \lg n)$ bits so that $P[i]$ and $P^{-1}[i]$ can be accessed in $O(f(n))$ time.

We store a bitvector A_1 of length at most $2n$ which for each apex (in order by the apex preorder numbers), has a 0 followed by i 1s, corresponding to the number of paths with

[5]This linear term can be optimized further, see Section 7.6

that apex. Finally, we store the paths for each apex explicitly using $n \lg n$ bits. Therefore to implement `apex_list` for an apex i we simply need to find the block of 1s by selecting for the i-th and $i+1$-st 0 in A_1 and retrieving the corresponding paths in $O(1)$ time per path. Thus `apex_list` can be implemented in $O(1)$ time per path.

Again we store T using $8n + o(n)$ bits.

Implementing `adjacent` Query: We are able to compute the pertinent features of the paths in $O(f(n))$ time, and thus `adjacent` can be done in $O(f(n))$ time.

Implementing `neighborhood` Query: As noted, we are able to implement `apex_list` in $O(1)$ time per path.

Case 1: We are able to handle the operation in $O(1)$ time per path.

Case 2: We will need to handle the 3-sided queries. To do so, we will need to be able to implement `rank` and `decode` for both x $(l(\cdot))$ and y $(r(\cdot))$ coordinates.

For `rank`, this is exactly the rank operation on L and R.

For `decode`, first consider the case that we need to compute the y-coordinate given an x rank. But this is exactly the case that given a vertex v, compute both its left and right end points $l(v), r(v)$, which we can accomplish in $O(1)$ time.

Conversely, suppose that we are given a y-coordinate rank r. We compute rank of the left endpoint using $P^{-1}[r]$. Once we have the rank of the left endpoint, we compute the preorder numbers of the left and right endpoints as above.

Thus we are able to implement the 2-D 3-sided queries using $O(f(n))$ per point.

Case 3: This is identical to the succinct data structure, except `apex_list` is now $O(1)$ per path.

We may summarize the result as:

Theorem 7.4.3. *Let G be a path graph. We may represent G using $(2 + \frac{1}{f(n)})n \lg n + o(n \lg n)$ bits to support* `adjacent` *in $O(1)$ time and* `neighborhood` *using $O(f(n))$ time per neighbour. We may also support* `neighborhood` *in $O(df(n))$ time.*

In particular, if $f(n) = \omega(1)$ then the leading term is $2n \lg n$. If $f(n) = \frac{1}{\varepsilon}$ for some constant $\varepsilon > 0$, then the space is $(2+\varepsilon)n \lg n$ and the time is $O(1)$.

Proof. Again `degree` follows from `neighborhood`. The total space required is:

- The tree T, the bitvectors L, R, A, Z, four 2-D 3-sided range reporting data structures from cases 2 and 3, stored succinctly: $O(n)$ bits[6].

[6]This linear term can be optimized further, see Section 7.6

- The permutation P using $(1+\frac{1}{f(n)})n\lg n + o(n\lg n)$ bits, using Lemma 2.3.8.
- The bitvector A_1 and the array storing the paths for `apex_list`: $n\lg n + 2n$ bits.

□

7.4.4 Distances

As path graphs are chordal graphs, we will use Lemma 7.3.1 to compute the distance. We note that to compute the distance, we needed a way to determine if there exists a w such that $w \in B(u_i) \cap B(v_j)$. Since G is now a path graph, if $w \in B(u_i) \cap B(v_j)$, then the two endpoints of w must be descendants of u_i and v_j as w must be a path. This can be expressed as a rectangle, stating that the left endpoint of w is a descendant of u_i and the right end point of w is a descendant of v_j: $[u_i, z_{u_i}] \times [v_j, z_{v_j}]$. As this is a four sided rectangle, we will use Theorem 7.4.2.

Theorem 7.4.4 (Corollary to Theorem 7.4.2 and Lemma 7.3.1)**.** *In addition to Theorem 7.4.2, we can answer the* `distance` *query in $O(\lg n/\lg\lg n)$ time and the* `spath` *query in $O(\lg n/\lg\lg n + d)$ time, where d is the distance between the two vertices. The extra space required for these two queries is $(1+\varepsilon)n\lg n + O(n)$ bits.*

Proof. We need to store the distance tree T_D succinctly, using $2n$ bits. As Munro and Wu [65] shown, there does not seem to be any way to easily associate a vertex in the graph and the corresponding node in T_D, and thus we must store a permutation and its inverse using $(1+\varepsilon)n\lg n$ bits using Lemma 2.3.8.

We compute the distance using level ancestor, and a single four sided query using the rectangle $[u_i, z_{u_i}] \times [v_i, z_{v_j}]$, taking $O(\lg n/\lg\lg n)$ time.

For shortest path, we move up the tree T_D using parent, and thus each vertex in the path takes $O(1)$ time to compute. We must again use $O(\lg n/\lg\lg n)$ time in our range query leading to an overall time complexity of $O(\lg n/\lg\lg n + d)$. □

7.5 Discussion

In this chapter, we gave better data structures for chordal graphs and path graphs than what is known in the literature.

For chordal graphs, we gave a more efficient implementation of `adjacent` and `decode`, reducing the time complexity from $O(g(n))$ for $g(n) = \omega(1)$ to $O(1)$. Since the data structure of Munro and Wu [65] uses these as the fundamental operations to implement everything, we were able to reduce the factor $g(n)$ for all of the queries.

For path graphs, we used orthogonal range search to realize a simpler global criteria to compute when two paths in a tree intersect. This lead to a succinct $n \lg n + o(n \lg n)$ bit data structure which supported the navigational queries in $O(\lg n / \lg \lg n)$ time rather than $O(\lg^2 n)$ required by Balakrishnan et al. [8] since they needed to examine $O(\lg n)$ interval graphs induced by a heavy-light decomposition of the tree. By exploiting the result on 2-dimensional 3-sided orthogonal range queries from Lemma 2.5.6, we were able to support the navigation queries in $O(1)$ time using just $2n \lg n + o(n \lg n)$ bits, where as Balakrishnan et al. needed $O(n \lg^2 n)$ bits.

Our results are not optimal, which leads to further research in this area. For chordal graphs, although we were able to reduce `adjacent` to $O(1)$, the approach for `neighborhood` requires a different approach to reduce it to $O(1)$ time. Both `distance` and `spath` use $O(n)$ time, which could be improved. However as Munro and Wu [65] showed, any such improvements would need to come from studying the set disjointment problem, which is thought to be difficult.

For path graphs, we gave two data structures which trades space efficiency for time efficiency: $n \lg n \to 2n \lg n$ space and $O(\lg n / \lg \lg n) \to O(1)$ time. A data structure that contains the best of both worlds $n \lg n$ space and $O(1)$ time would be desirable.

7.6 Optimizations

Here we give some small optimizations in the lower order terms of our data structures. Specifically, the linear terms in our $n \lg n$ bit space data structures.

First is the optimization for our 2-dimensional 3-sided range search data structure.

We may reduce the space required for the range minimum/maximum data structure by first reporting an approximate minimum. First divide the array into blocks of size h and denote the value of each block as the minimum/maximum of the values in the block. Thus we have a condensed array of size n/h. We build the range minimum/maximum structure on the condensed array. For any query, we reduce it to a query on the condensed array, and linearly search the block of the minimum/maximum value. There are two incomplete blocks that we need to linearly search as well. In this manner, and setting $h = 2/\varepsilon$ we obtain space equal to εn bits and time equal to $O(f + k \cdot g)$ for any constant $\varepsilon > 0$.

Next is the bit vectors L, R, A, Z of section 7.4. As every node in the tree can be exactly one of the above categories (left endpoint, right endpoint, apex, last node) we may represent them instead using a single string of length $4n$, where at each index i, the string stores a digit between 0 and 4 indicating which category the node i belongs to (with 4 being none of the above).

We may store this using Lemma 2.3.6, using space equal to $(4 \lg 5)n$ bits instead of $16n$ bits. Furthermore, we are guaranteed, exactly n 0s and 1s corresponding to the left and right end points of the paths. To maximize the entropy, we would like to have $2n/3$ nodes of each of the other 3 categories. This gives a 0-th order entropy of $1 + \frac{\lg 6}{2} \approx 2.29$ versus $\lg 5 \approx 2.32$. Thus we may further compress these bitvectors down to $4(1 + \frac{\lg 6}{2})n$ bits using the result of [10].

Lastly, the tree T of section 7.4 has many leaves, since any node that is the left end point or right end point of a path and the last node of subtrees (i.e. z_v) are all leaves by construction. The maximum ratio of leaves to internal nodes is thus 3 leaves for each internal node. Using the result of [50], we may represent the tree using $3n \lg(4/3) + n \lg(1/4) + 2n \leq 5.25n$ bits, rather than $8n$ bits.

References

[1] Ittai Abraham and Cyril Gavoille. On approximate distance labels and routing schemes with affine stretch. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 404–415. Springer, 2011.

[2] 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.

[3] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. 1963.

[4] Josh Alman, Timothy Chu, Aaron Schild, and Zhao Song. Algorithms and hardness for linear algebra on geometric graphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 541–552. IEEE, 2020.

[5] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees. *ACM Transactions on Algorithms*, 1, 12 2003.

[6] 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.

[7] Joyce Bacic, Saeed Mehrabi, and Michiel Smid. Shortest beer path queries in outerplanar graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka,*

Japan, volume 212 of *LIPIcs*, pages 62:1–62:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] Girish Balakrishnan, N. S. Narayanaswamy, Sankardeep Chakraborty, and Kunihiko Sadakane. Succinct data structure for path graphs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 262–271. IEEE, 2022.

[9] 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.

[10] 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.

[11] Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, June 2004.

[12] 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.

[13] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

[14] 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.

[15] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

[16] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *WADS*, 2009.

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

[18] Sankardeep Chakraborty and Seungbum Jo. Compact representation of interval graphs of bounded degree and chromatic number. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 103–112. IEEE, 2022.

[19] Sankardeep Chakraborty, Seungbum Jo, Kunihiko Sadakane, and Srinivasa Rao Satti. Succinct data structures for series-parallel, block-cactus and 3-leaf power graphs. In Ding-Zhu Du, Donglei Du, Chenchen Wu, and Dachuan Xu, editors, *Combinatorial Optimization and Applications - 15th International Conference, COCOA 2021, Tianjin, China, December 17-19, 2021, Proceedings*, volume 13135 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2021.

[20] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011.

[21] Shiri Chechik. Approximate distance oracles with improved bounds. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 1–10. ACM, 2015.

[22] Jingbang Chen, Meng He, J. Ian Munro, Richard Peng, Kaiyu Wu, and Daniel Zhang. Distance queries over dynamic interval graphs.

[23] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.

[24] 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.

[25] 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.

[26] Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, Anurag Murty Naredla, and Kaiyu Wu. Shortest beer path queries in interval graphs. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, pages 59:1–59:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[27] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the louds succinct tree representation. In Carme Àlvarez and María Serna, editors, *Experimental Algorithms*, pages 134–145, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] Hicham El-Zein, Moshe Lewenstein, J Ian Munro, Venkatesh Raman, and Timothy M Chan. On the succinct representation of equivalence classes. *Algorithmica*, 78:1020–1040, 2017.

[29] Arash Farzan and J. Ian Munro. Succinct representation of finite abelian groups. In Barry M. Trager, editor, *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9-12, 2006, Proceedings*, pages 87–92. ACM, 2006.

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

[31] Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, June 2014.

[32] 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.

[33] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 1–7, New York, NY, USA, 1990. Association for Computing Machinery.

[34] Delbert Ray Fulkerson and Oliver Alfred Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[35] 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.

[36] Fănică Gavril. A recognition algorithm for the intersection graphs of paths in trees. *Discrete Mathematics*, 23(3):211–227, 1978.

[37] 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.

[38] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, pages 326–337, Cham, 2014. Springer International Publishing.

[39] Alexander Golynski, J. Munro, and Srinivasa Rao Satti. Rank/select operations on large alphabets: a tool for text indexing. pages 368–373, 01 2006.

[40] G Hajós. Über eine art von graphen. int. *Math. Nachr*, 11:1607–1620, 1957.

[41] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent Advances in Fully Dynamic Graph Algorithms. In James Aspnes and Othon Michail, editors, *1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022)*, volume 221 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:47, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[42] Phil Hanlon. Counting interval graphs. *Transactions of the American Mathematical Society*, 272(2):383–383, February 1982.

[43] Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[44] 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.

[45] Meng He, J. Ian Munro, and Kaiyu Wu. Succinct intersection graphs revisited.

[46] Monika Henzinger. The state of the art in dynamic graph algorithms. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 40–44, Cham, 2018. Springer International Publishing.

[47] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, jul 1999.

[48] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $o(\log n(\log\log n)^2)$ amortized expected time. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 510–520. SIAM, 2017. available at https://arxiv.org/abs/1609.05867.

[49] 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.

[50] 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.

[51] 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.

[52] Pavel Klavík and Peter Zeman. Automorphism Groups of Geometrically Represented Graphs. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 540–553, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[53] Johannes Köbler, Sebastian Kuhnert, Bastian Laubner, and Oleg Verbitsky. Interval graphs: Canonical representation in logspace. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming*, pages 384–395, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[54] Hung Le and Christian Wulff-Nilsen. Optimal approximate distance oracle for planar graphs. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 363–374. IEEE, 2021.

[55] Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2517–2537. SIAM, 2021.

[56] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4:28:1–28:13, July 2008.

[57] S. Melczer. *An Invitation to Analytic Combinatorics: From One to Several Variables*. Texts & Monographs in Symbolic Computation. Springer Nature Switzerland, 2020.

[58] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on ram. *SIAM Journal on Computing*, 35(6):1494–1525, 2006.

[59] J. Ian Munro and Patrick K. Nicholson. Succinct posets. *Algorithmica*, 76(2):445–473, 2016.

[60] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao S. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[61] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, January 2001.

[62] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[63] 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.

[64] 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.

[65] 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.

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

[67] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):1–39, may 2014.

[68] Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1191–1205. SIAM, 2021.

[69] Mihai Patrascu. Succincter. In *Symposium on Foundations of Computer Science (FOCS)*. IEEE, October 2008.

[70] Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014.

[71] G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68(none):145 – 254, 1937.

[72] 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.

[73] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[74] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[75] Siddhartha Sen, Robert E. Tarjan, and David Hong Kyun Kim. Deletion without rebalancing in binary search trees. *ACM Trans. Algorithms*, 12(4), sep 2016.

[76] 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.

[77] N. Sloane. The on-line encyclopedia of integer sequences. *Notices Amer. Math. Soc.*, 50:912–915, 09 2003.

[78] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, April 2014.

[79] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[80] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[81] Konstantinos Tsakalidis, Sebastian Wild, and Viktor Zamaraev. Succinct permutation graphs. *CoRR*, abs/2010.04108, 2020.

[82] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[84] 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.

[85] 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.

APPENDICES

Appendix A

List of Definitions

Definition 2.1.1. A (undirected) ***graph*** G consists of a set of vertices $V(G)$ and a set of edges $E(G) \subset V(G)^2$, and if needed write $G = (V(G), E(G))$. When the graph is unambiguous, we will use V and E. We will use $n = |V|$ and $m = |E|$.

Definition 2.1.2. A graph G is ***weighted*** if we assign a real numbered weight to each edge. If all weights are 1, we say that the graph is ***unweighted***.

Definition 2.1.3. The operations we are interested in supporting in a graph data structure are:

- `adjacent`(u, v)[1]: given two vertices, are they adjacent (i.e. is $(u, v) \in E$)?
- `degree`(v): given a vertex, what is the number of vertices adjacent to it?
- `neighborhood`(v): given a vertex, list all vertices adjacent to it.
- `spath`(u, v): given two vertices, return a shortest (weighted) path between them.
- `distance`(u, v): given two vertices, return the length of a shortest path.

The operations below modify the graph. A graph data structure supporting these are *dynamic*.

- `insert`(v): add a new vertex v to the graph

[1]Our data structures will have their own ways to referring of vertices, typically by giving each vertex a unique label between 1 and n, and the inputs to our operations will be the label of that vertex.

- `insert`(u, v): given two vertices in the graph, add an edge between them.
- `delete`(v): delete the vertex v in the graph
- `delete`(u, v): delete the edge between u, v in the graph.

Definition 2.1.4. A graph G is an ***intersection graph*** if we may associate every vertex v with a set s_v such that for any two vertices, $(u, v) \in E$ if an only if $s_u \cap s_v \neq \emptyset$. We say that the family of sets is an ***intersection model*** for the graph.

Definition 2.1.5. A rooted ***ordinal tree*** is an acyclic graph consisting of n vertices (which we will call *nodes*) and $n - 1$ edges, with one node designated as the root. All non-root vertices have a parent vertex, which is closer in distance to the root. The *children* of a node v are the nodes whose parent is v. A node is a *leaf* if it has no children. We note that for an ordinal tree, the order of the children of a node matters (i.e. two ordinal trees are isomorphic if for corresponding nodes, the children are the same and have the same order).

Definition 2.1.6. A rooted ***cardinal tree*** (of cardinality k) is an ordinal tree where for every node, each child belongs to one of k slots, and no slot is used more than once.

Definition 2.1.7. A ***binary tree*** is an cardinal tree of cardinality 2, where we name the slots as "left" and "right". Thus, each child of a node v is either a left child or a right child. Consequently, if a node has a single child, then the 2 trees where the child is a left child versus a right child are different trees.

Definition 2.1.8. A graph G is a ***chordal graph*** if it is the intersection graph of subtrees in a tree. That is, there exists an ordinal tree T, such that for every vertex $v \in V$, s_v is a set of connected nodes in T.

Definition 2.1.9. A graph G is a ***path graph*** [2] if it is the intersection graph of (simple) paths in a tree. That is, there exists an ordinal tree T, such that for every vertex v, s_v is a set containing the nodes of a simple path in T.

Definition 2.1.10. A graph G is an ***interval graph*** if it is the intersection graph of (simple) paths in a path. That is, there exists an ordinal tree T that is a path, such that for every vertex v, s_v is a set containing the nodes of a simple path in T.

[2]The object we are defining is not a path, which is what is often referred to by the term path graph.

Definition 2.2.1. A data structure for a class of combinatorial objects X is ***succinct*** if it uses $\lg|X| + o(\lg|X|)$ bits in the worst case. It is ***compact*** if it uses $\Theta(\lg|X|)$ bits in the worst case.

The data structure will naturally need to support the relevant queries on X. [3]

Definition 2.3.1. A ***bit-vector*** of length n is an array of n bits. It supports the following operations:

- `access`(i): return the bit at index i.
- `rank`$_b(i)$: return the number of "b" bits at or before the index i, where b is either 0 or 1. If we omit b, it is assumed that we mean $b = 1$.
- `select`$_b(i)$: return the index of the i-th "b" bit. If we omit b, it is assumed that we mean $b = 1$.

Definition 2.3.5. A ***string*** S of length n on an alphabet $\Sigma = \{1, \ldots, \sigma\}$ is a sequence n characters belonging to Σ. The operations generalize naturally as:

- `access`(i): return the character at index i.
- `rank`(c, i): return the number of occurrences of the character c in the prefix up to (and including) the index i.
- `select`(c, i): return the index of the i-th occurrence of the character c.

Definition 2.3.7. A ***permutation*** P of size n is a bijective function from $\{1, \ldots, n\}$ to itself. The two operations are:

- $P[i]$: return the value that i is sent to by the function.
- $P^{-1}[i]$: return the value that is sent to i by the function.

Definition 2.4.3. Let A be an array of numbers. A ***range-minimum*** query is the following:

- Given two indices i, j, return the index k, $i \le k \le j$ such that $A[k] = \min_{i \le k \le j} A[k]$ (i.e. the index between i and j containing the minimum element in that range)

[3]One can naively represent the objects by simply enumerating them and giving them numbers $1, \ldots, |X|$. Such a scheme is not a *data structure* since we cannot answer queries - such as `adjacent` for a graph.

Definition 2.5.1. Let S be a set of n d-dimensional points (with polynomial sized coordinates). Let R be a d-dimensional axis aligned rectangle (given as input to a query): $[p_1, p_2] \times [p_3, p_4] \dots [p_{2d-1}, p_{2d}]$ [4]. The queries we wish to support are:

- *emptiness*: is there a point of S inside R?
- *count*: how many points of S are inside R.
- *reporting*: return every point of S inside R.

The rectangle R is k-sided if there is at most k coordinates among p_i that are finite.

Definition 2.5.4. The data structure stores a set of n numbers S out of a universe $U = [1, |U|]$. The queries are:

- `pred`(i), given an element i of U, return the largest element j of S that is smaller than i.
- `succ`(i), given an element i of U, return the smallest element j of S that is larger than i.

Definition 3.3.1. Let A be an array storing values of k bits. We say that A is piece-wise linear with M pieces if there are M runs of identical values.

Definition 3.4.3. The parameters for our tree-slabbing scheme are $H = \lceil \lg^3 n \rceil$ and $B = \lceil \lg^5 n \rceil$ and $H' = \lceil \frac{\lg n}{(\lg \lg n)^2} \rceil$ and $B' = \lceil \frac{1}{8} \lg n \rceil$

Definition 3.4.4. A node v so that $\tau_1(v) \neq \tau_1(v-1)$ is called a ***(tier-1) preorder changer*** [44, Def. 4.1]. Similarly, nodes v with $\tau_2(v) \neq \tau_2(v-1)$ are called ***(tier-2) preorder changers***.

Definition 3.4.6. 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.

Definition 4.3.1. A graph G is an ***interval graph*** if it is the intersection graph of intervals on the real line. That is for every vertex v, s_v is a set containing the real numbers in the interval $[l_v, r_v]$. In this way, we will name the interval $I_v = [l_v, r_v]$.

[4]We use closed interval here, but it is easy to see how to convert these into open/semi-open intervals when the coordinates are integral

Definition 4.3.2. A ***proper interval graph*** is an interval graph where there exists a way to associate each vertex v with an interval $I_v = [l_v, r_v]$ such that no two interval nest. That is for all u, v, $I_v \cap I_u \notin \{I_v, I_u\}$.

Definition 4.3.3. An interval graph is ***k-proper*** if there exists an way to associate each vertex v with an interval such that for every vertex v, the number of intervals containing it is at most k. An interval graph is ***k-improper*** if there exists an way to associate each vertex v with an interval such that for every vertex v, the number of intervals it contains is at most k.

Definition 4.3.5. An interval graph is a ***bounded degree interval graph*** with respect to a parameter σ if the maximum degree over all vertices is σ. An interval graph is a ***bounded chromatic number interval graph*** with respect to a parameter σ if we may assigned a colour $c_v \in [1, \ldots, \sigma]$ to each vertex such that any two adjacent vertices have different colours.

Definition 4.3.6. A graph is a ***circular arc graph*** if it is the intersection graph of arcs on a circle.

Definition 5.2.1. A ***beer graph*** is a tuple (G, B) consisting of a graph and a set of beer vertices $B \subseteq V$.

Definition 5.2.2. If (G, B) is a beer graph, then we are interested in these queries:

- `beer_spath` (u, v): return a shortest path between the vertices u and v such that at least one of the beer vertices appears on the path.
- `beer_distance` (u, v): return the length of the shortest path between vertices u and v such that at least one of the beer vertices appears on the path.

Definition 5.3.2. Given two vertices u, v, a vertex w with $u < w < v$ ***preserves the distance*** (w.r.t. u, v) if $\texttt{distance}(u, v) = \texttt{distance}(u, w) + \texttt{distance}(w, v)$.

Definition 5.4.1. Given two vertices u, v, for a node w (such that $u < w < v$), we say that w is ***+k the distance*** (w.r.t. u, v) if $\texttt{distance}(u, v) + k = \texttt{distance}(u, w) + \texttt{distance}(v, w)$.

Definition 6.2.1. Let G be an interval graph, with a fixed interval representation. The ***distance tree*** $T(G)$ is defined under the parent relationship $\texttt{parent}(v)$. For every vertex v, we order the children of v in order of the left end point of the vertices. That is, if u, w are two children of v with $l_u < l_w$ then u is to the left of w.

If the graph is disconnected, then we will have a forest instead. We have one tree per vertex v where $\texttt{parent}(v) = v$. Furthermore when we refer to the distance tree $T(G)$ of G in the context of a vertex v if G is disconnected, it is understood that we refer to the tree in the forest that contains v.

Definition 6.5.1. For an interval graph G with intervals $\mathcal{I}$, we say that an interval/vertex $x \in \mathcal{I}$ is ***exposed*** if it is not contained by another interval of $\mathcal{I}$, and we let $\mathcal{I}^{\texttt{exposed}}(G)$ denote the set of all exposed interval of G.

Definition 6.6.1. Let G be an interval graph with a fixed interval representation $\mathcal{I}$, and let S be a set of points. We will say that $R(G, S)$ is an ***reduced representation*** of G if:

- for any two intervals $u = [l_u, r_u], v = [l_v, r_v]$ with endpoints in S, we may compute the query
$$\texttt{distance}_{G\cup\{u,v\}}(u, v)$$
using $R(G, S)$
- If $\mathcal{I}_S$ is any set of intervals with endpoints in S, then we are able to compute $R(G', S)$ from $R(G, S)$ only (that is without requiring knowledge of G) for $G' = G \cup \mathcal{I}_S$. We will define this operation as $R(G', S) = \texttt{batch\_insert}(R(G, S), \mathcal{I}_S)$.
- If $S' \subset S$, then we are able to compute $R(G, S')$ from $R(G, S)$. We will define this operation as $R(G, S') = \texttt{reduce\_active}(R(G, S), S')$.

Definition 6.6.5. For the distance tree $T_{(\hat{G},S)}$, we say that a vertex is ***active*** if it is the vertex corresponding to a point in S or it is a vertex v such that $l_v \in S$. (Note that by Definition 6.6.4, we insert all points in S that is not the left end point of any interval in $\hat{G}$)

Definition 6.6.6. Define the ***compressed distance tree*** $\hat{T}_{(\hat{G},S)}$ by compressing all paths in $T_{(\hat{G},S)}$ by keeping only active vertices and any lowest common ancestors of active vertices. $\hat{T}$ is a weighted tree where the edges are weighted by the lengths of the compressed paths.

Definition 6.7.1. Let v be a vertex in G, we say that v has an ***in-block*** parent if v and $\texttt{parent}_T(v)$ are in the same block (or if v is the root of the distance tree of its component) and v has an ***out-of-block*** parent otherwise.