

Distance Queries over Dynamic Interval Graphs

Jingbang Chen ✉ 🏠 

Cheriton School of Computer Science, University of Waterloo, Canada

Meng He ✉ 🏠 

Faculty of Computer Science, Dalhousie University, Halifax, Canada

J. Ian Munro ✉ 🏠 


Cheriton School of Computer Science, University of Waterloo, Canada

Richard Peng ✉ 🏠

Cheriton School of Computer Science, University of Waterloo, Canada

Kaiyu Wu ✉ 

Cheriton School of Computer Science, University of Waterloo, Canada

Daniel J. Zhang ✉ 

School of Computer Science, Georgia Tech, Atlanta, GA, USA

Abstract

We design the first dynamic distance oracles for interval graphs, which are intersection graphs of a set of intervals on the real line, and for proper interval graphs, which are intersection graphs of a set of intervals in which no interval is properly contained in another.

For proper interval graphs, we design a linear space data structure which supports distance queries (computing the distance between two query vertices) and vertex insertion or deletion in $O(\lg n)$ worst-case time, where n is the number of vertices currently in G . Under incremental (insertion only) or decremental (deletion only) settings, we design linear space data structures that support distance queries in $O(\lg n)$ worst-case time and vertex insertion or deletion in $O(\lg n)$ amortized time, where n is the maximum number of vertices in the graph. Under fully dynamic settings, we design a data structure that represents an interval graph G in $O(n)$ words of space to support distance queries in $O(n \lg n / S(n))$ worst-case time and vertex insertion or deletion in $O(S(n) + \lg n)$ worst-case time, where n is the number of vertices currently in G and $S(n)$ is an arbitrary function that satisfies $S(n) = \Omega(1)$ and $S(n) = O(n)$. This implies an $O(n)$ -word solution with $O(\sqrt{n \lg n})$ -time support for both distance queries and updates. All four data structures can answer shortest path queries by reporting the vertices in the shortest path between two query vertices in $O(\lg n)$ worst-case time per vertex.

We also study the hardness of supporting distance queries under updates over an intersection graph of 3D axis-aligned line segments, which generalizes our problem to 3D. Finally, we solve the problem of computing the diameter of a dynamic connected interval graph.

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

Keywords and phrases interval graph, proper interval graph, intersection graph, geometric intersection graph, distance oracle, distance query, shortest path query, dynamic graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2023.18

Related Version *Thesis Ch. 6*: <https://uwspace.uwaterloo.ca/handle/10012/19686>

1 Introduction

The computation of the shortest path and the distance between a pair of vertices in a graph are fundamental problems in graph algorithms. The *shortest path* between two vertices x and y in an unweighted graph is the path with the fewest number of edges with x and y as endpoints, and the *distance* between x and y is the number of edges in this path.



© Jingbang Chen, Meng He, J. Ian Munro, Richard Peng, Kaiyu Wu, and Daniel J. Zhang; licensed under Creative Commons License CC-BY 4.0

34th International Symposium on Algorithms and Computation (ISAAC 2023).

Editors: Satoru Iwata and Naonori Kakimura; Article No. 18; pp. 18:1–18:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The study of these problems has many applications including contextual searching [27], social network analysis [28] and molecular topology indices [29].

To support online queries, *shortest path oracles* and *distance oracles* have been designed. They are constructed by preprocessing the given n -vertex graph G , such that, given a pair of 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, but the space cost is quadratic. As this space cost is believed to be necessary for fast distance queries, to improve space efficiency, much work has been done to design approximate distance oracles [24, 1]. For example, given a pair of vertices x and y at distance d , the $O(n^{5/3})$ -word distance oracles of Pătraşcu and Roditty [24] can compute in constant time an approximate distance in $[d, 2d + 1]$.

These distance oracles often use $O(n^{1+\Omega(1)})$ space, so for modern applications processing large graphs, they 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 [21, 20] and potentially dense graphs such as interval graphs [17, 18] and chordal graphs [25, 23]. Among them, *interval graphs* are intersection graphs of a set of intervals on the real line and have applications in operations research [5] and bioinformatics [30].

The recent result of He et al. [18] is a succinct representation of interval graphs that occupies $n \lg n + (5 + \epsilon)n + o(n)$ bits of space (which is $n + o(n)$ words) for any constant positive ϵ and answers distance queries in constant time. It can also answer shortest path queries using time linear in the length of the path. In addition, they show how to represent a *proper interval graph*, which are interval graphs with no interval properly contained in another, in $2n + o(n)$ bits to provide the same query support.

One reason why the above work on interval graphs 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 of 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 linear or near linear space or algorithms that work in linear or near linear time, without explicitly constructing all edges. Other instances include the work of Alman et al. [3] on geometric graphs and that of Munro and Sinnamoni [22] on distributive lattices.

Previous work on distance oracles for interval graphs focused on static graphs, while distance oracles for dynamic general graphs require $\Omega(n^2)$ update time [14, 26]. Previous results on dynamic interval graph data structures, worked under the update model of inserting and deleting individual edges, with update times $\Theta(n)$ [13]. Hence, in this paper, we design data structures that support distance and shortest path queries over dynamic interval and proper interval graphs, under the model of updating the graph by the insertion and deletion of an interval, along with all the corresponding edges. By working on these problems, 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?”

Previous Work. To solve the all-pairs shortest paths problem over interval graphs, Chen et al. [11] built an $O(n)$ -word structure that answers distance queries in $O(1)$ times. Further work by Acan et. al [2] in reducing the space yielded $n \lg n + (3 + \epsilon)n + o(n)$ (for constant

$\varepsilon > 0$) bits of space data structure for shortest paths, and further improvements of He et al. [18] gave a $n \lg n + (5 + \varepsilon)n + o(n)$ bits of space data structure that also allows distance queries in $O(1)$ time. In the same work, Acan et al. [2] showed how to use $2n + o(n)$ bits of space to support shortest path queries in proper interval graphs in $O(1)$ time per vertex on the path, and He et al. [18] showed how to support distance queries in the same $2n + o(n)$ bits of space, in $O(1)$ time ¹.

In a slightly different model of distance labeling, the data structure is distributed among the vertices, and we compute the distance between two vertices from only their labels. The best known result is a $5 \lg n$ -bit label of Gavaille and Paul [17] (so a total space of $5n \lg n$ bits), which computes the distance in $O(1)$ time.

Interval graphs are a subset of circular arc graphs, which are intersection graphs of arcs on a circle. The results on interval graphs can be extended to circular arc graphs by unrolling the circular arc graph (twice) and reducing it to an interval graph instance on twice the number of vertices. Interval graphs are also a subset of chordal graphs, which are intersection graphs of subtrees in a tree. For chordal graphs, we have the approximate distance oracle of Singh et. al [25] which uses $O(n)$ words of space and computes in $O(1)$ time a distance between $[d, 2d + 8]$ where d is the true distance. This was improved by Munro and Wu [23] to compute a distance between $[d, d + 1]$ using $n + o(n)$ words. Munro and Wu also gave an exact oracle using $n^2/4 + o(n^2)$ bits of space with query time $O(nf(n))$ for any $f(n) \in \omega(1)$.

Another way of generalizing interval graphs is to define intersection graphs of line segments or boxes in two or higher dimensions. Chan and Skrepetosz [9] solved the all-pairs shortest path problem over several classes of geometric intersection graphs, including intersection graphs of axis-aligned line segments, arbitrary line segments or axis-aligned boxes. Researchers have also designed distance oracles for intersection graphs over unit disks [16, 10].

Our Results. We consider interval graphs given as a set of intervals with edges represented implicitly by the intersections between these intervals. An update is performed by adding a vertex represented by an interval, and this implies the insertions of all its incident edges implicitly, or by deleting vertices represented by an interval which implies the removal of all its incident edges. This is natural in our setting as our edges are implicit in this intersection model, and adding or deleting arbitrary edges may give us a graph outside of our graph class.

Under this model, we design a data structure in Section 3 that represents a proper interval graph G in $O(n)$ words, where n is the number of vertices currently in G , to answer distance queries and to support vertex insertion or deletion in $O(\lg n)$ worst-case time. The shortest path query can also be answered in $O(\lg n)$ worst-case time per vertex on the path.

For general interval graphs, we first consider the incremental case, in which we start from an empty graph and insert n vertices one by one, and the decremental case, in which we start from an n -vertex graph and perform n vertex deletions.

Queries can be made at any time during these update sequences. For these settings, we design in Section 4 an $O(n)$ -word representation that supports distance queries in $O(\lg n)$ worst-case time, and vertex insertions (in the incremental case) or deletions (in the decremental case) in $O(\lg n)$ amortized time. The shortest path query can be answered in $O(\lg n)$ worst-case time per vertex on the path.

¹ Although their data structure uses $3n + o(n)$ bits if the graph is not connected, this is due to using n additional bits to determine if vertices belong to the same component. A $O(\sqrt{n})$ additional bit solution to detect components can be found using the equivalence class data structure of El-Zein et al [15].

Under fully dynamic settings in which we can mix insertions and deletions, we further design in Section 5 a data structure that represents a general interval graph G in $O(n)$ words of space to answer distance queries in $O(n \lg n / S(n))$ worst-case time and to support the insertion or deletion of a vertex into or from G in $O(S(n) + \lg n)$ worst-case times, where n is the number of vertices currently in G and $S(n)$ is an arbitrary function that satisfies $S(n) = \Omega(1)$ and $S(n) = O(n)$. It also answers a shortest path query in $O(\lg n)$ worst-case time per vertex on the path. Thus, setting $S(n) = \sqrt{n \lg n}$ yields an $O(n)$ -word solution with $O(\sqrt{n \lg n})$ -time support for both distance queries and updates. These solutions are the first that support distance and shortest path queries over dynamic interval and proper interval graphs efficiently.

In addition, we also study in Section 6 the hardness of the problem of supporting distance queries under updates in an intersection graph of 3D axis-aligned line segments, which generalizes our problem to 3D. We reduce the online Boolean Matrix vector multiplication (OMv) problem [19] to it. Thus, for any constant $\varepsilon > 0$, the distance query and update times over such a graph cannot be $O(n^{1/2-\varepsilon})$ simultaneously, unless the OMv conjecture [19] is false. This implies conditional lower bounds for more general graphs such as intersection graphs of 3D axis-aligned boxes and intersection graphs of arbitrary line segments in 3D [9, 7, 12, 8].

Due to space constraints, some details are omitted.

2 Definitions and Preliminaries

2.1 Definitions

Let $G = (V, E)$ denote a graph with vertex set V and edge set E , and we consider unweighted graphs only. We use $n = |V|$ and $m = |E|$ to refer to the number of vertices and edges, respectively. As is standard, we assume the word-RAM model with $\Theta(\lg n)$ -size words.

An *intersection graph* is formed from a finite family of non-empty sets. We associate each vertex with a set, and two vertices are adjacent if and only if the corresponding sets intersect. Then, an *interval graph* is an intersection graph of a set of intervals on the real line, while 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.

The representing interval of a vertex v of an interval graph is denoted by $I_v = [l_v, r_v]$. We use \mathcal{I} to refer to the current interval set of the graph, and we say that \mathcal{I} is an *interval representation* of G . \mathcal{I} may change when the graph is dynamic. We define the following operators over an interval graph G :

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

2.2 Static Data Structure for Distances in Interval Graphs

Here we will review some previous results for the static case [18, 2], which we will build upon.

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

$$u = \arg \min \{l_w \mid r_w \geq l_v\} \quad (1)$$

► **Definition 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 G is disconnected, then we have a forest instead, with one tree per vertex v where $\text{parent}(v) = v$. Furthermore, in the context of a vertex v , the distance tree $T(G)$ of a disconnected graph G refers to the tree in the forest that contains v .

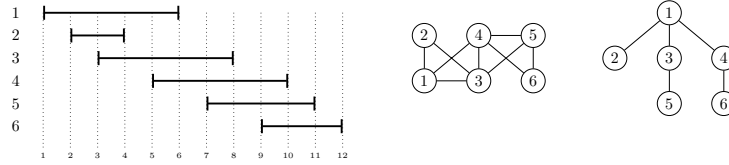


Figure 1 An Interval Graph (middle) with Interval Representation (left), and distance tree constructed (right).

For a tree T and a node v , the quantity $\text{pos}_T^X(v)$ denotes the index of v in the X traversal of T , where X could be *level*, *pre*, *post* indicating a breadth-first traversal, preorder traversal and postorder traversal, respectively. We will omit the subscript when the tree being referred to is clear. Then we have

► **Lemma 2** (Lemma 7 of [18]). *Let G be a proper interval graph with distance tree $T(G)$ and vertices u, v . $\text{pos}^{\text{level}}(u) < \text{pos}^{\text{level}}(v)$ if and only if $l_u < l_v$. In the special case that $\text{depth}(u) = \text{depth}(v)$, $\text{pos}^X(u) < \text{pos}^X(v)$ if and only if $l_u < l_v$ for $X = \text{pre}, \text{post}, \text{level}$.*

The main property of the distance tree is that it encodes distances between vertices.

► **Lemma 3.** *Let G be an interval graph with distance tree $T(G)$, and u, v be two vertices in the same connected component of G with $\text{pos}^{\text{level}}(u) > \text{pos}^{\text{level}}(v)$. Let the node to root path of u be $u = u_1, \dots, u_k = r$, and i be the first index where $l_{u_i} \leq r_v$. Then a shortest path from u to v is $u = u_1, \dots, u_i, v$, and $\text{depth}(u_i)$ is $\text{depth}(v) - 1$, $\text{depth}(v)$ or $\text{depth}(v) + 1$.*

In the proper interval graph case, we may state it more succinctly as

► **Lemma 4.** *Let G be a proper interval graph with distance tree $T(G)$ and u, v be two vertices in the same connected component of G with $\text{pos}^{\text{level}}(u) > \text{pos}^{\text{level}}(v)$. Then $\text{dist}(u, v) = \text{depth}(u) - \text{depth}(v) + \mathbf{1}(\text{pos}^{\text{post}}(u) > \text{pos}^{\text{post}}(v))$, where the last term evaluates to 1 if the expression inside the brackets is true and 0 otherwise.*

3 Fully Dynamic Proper Interval Graphs

As proper interval graphs are a special class of interval graphs, we naturally modify the $\text{insert}(v)$ 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 abort the operation.

Our dynamic solution modifies the distance tree of He et al. [18] so that it is more easily maintainable. The distance tree $T(G)$ (Definition 1) in general has an unbounded degree, which makes updates difficult. 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 depths, sibling edges in the binary tree will have a weight 0, while parent edges in the binary tree will have a weight 1.

More formally, let T be an ordinal tree on n nodes. Define the weighted binary tree 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 rightmost child in T . Left child edges have a weight 0, right child edges have a 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 invariant under this transformation.

Thus Lemma 4 still holds under this transformation. Furthermore, we will use $\text{pos}^{\text{level}}(u)$ 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 $u = \text{anc}_T(v, d)$ 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 (as the edges may have 0 weights, there are multiple ancestors at each depth). We will store the tree $T_B(G)$ as a top tree [4], which allows us to make dynamic changes along with useful path queries in $O(\lg n)$ time.

► **Lemma 5** (Top Tree [4]). *Let T be a forest. A top tree data structure on T occupies $O(n)$ words of space and supports the following operations in $O(\lg n)$ time: $\text{link}(u, v)$, where u and v are in different trees, links these trees by adding the edge (u, v) to T ; $\text{cut}(e)$, removes the edge e from T ; $\text{update_weight}(e, w)$, update the weight of the edge e to w ; $\text{weighted_distance}(u, v)$, returns the weight of the path between u and v ; $\text{anc}(u, v, d)$, returns the first node on the path from u to v at distance d from u .*

For a proper interval graph G , with intervals \mathcal{I} , we will maintain the following: **1)** A top tree of $T_B(G)$. For each component, we store a variable indicating the root r of that component. **2)** A mapping between the vertices v of G and the interval $I_v = [l_v, r_v]$. **3)** 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. **4)** The left end points of all the intervals. **5)** The right end points of all the intervals. For the last 4 items, we will use a red-black tree, 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 $\text{pred}_L/\text{succ}_L$, on the set of left endpoints and $\text{pred}_R/\text{succ}_R$ on the set of right endpoints. The total space is $O(n)$ words.

Now we translate the distance calculation from $T(G)$ into our data structure essentially by translating the comparison between $\text{pos}^{\text{level}}$ to a comparison between interval endpoints.

► **Lemma 6.** *The data structures in this section can compute $\text{dist}(u, v)$ in $O(\lg n)$ time given two vertices, u, v of G .*

Proof. In the case that $\text{depth}(u) = \text{depth}(v)$, we retrieve the endpoints. Assume without loss of generality that $l_u > l_v$. Since $\text{pos}^{\text{post}}(u) > \text{pos}^{\text{post}}(v) \Leftrightarrow \text{pos}^{\text{level}}(u) > \text{pos}^{\text{level}}(v) \Leftrightarrow l_u > l_v$, by lemma 4 we have that $\text{dist}(u, v) = 1$.

Now suppose that $\text{depth}(u) > \text{depth}(v)$. By the property of a breadth-first traversal, we also have $\text{pos}^{\text{level}}(u) > \text{pos}^{\text{level}}(v)$. Using the top tree, we find $w = \text{anc}(u, \text{depth}(v))$ in $O(\lg n)$ time. Then $\text{pos}^{\text{post}}(u) > \text{pos}^{\text{post}}(v) \Leftrightarrow \text{pos}^{\text{post}}(w) > \text{pos}^{\text{post}}(v) \Leftrightarrow l_w > l_v$, so a comparison between l_w and l_v is sufficient to apply lemma 4. ◀

Now we consider the maintaining of 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 7.** *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 $\text{parent}_{T_B(G)}(v)$ is*

$$\arg \min (\{l_w \mid l_w \geq l_v\} \cup \{r_w \mid r_w \geq l_v\}) \quad (2)$$

If two vertices u, w have endpoints such that $r_u = l_w$, break ties in the above quantity by treating $l_w < r_u$. Furthermore, let v' be the vertex such that $l_{v'} = \text{pred}_L(l_v)$. 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 come 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 $l_u = \text{succ}_L(l_v)$, u must be the immediate 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 note that v is the rightmost child of u in $T(G)$. This can be seen as there are no vertices v' with $l_v \leq l_{v'} \leq r_u$ which any right sibling of v must have. ◀

To support updates, 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 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 successor of w .

► **Lemma 8.** *Let G be a proper interval graph with intervals \mathcal{I} . Let $w = [l, r]$ such that $l \neq l_v$ is² not the left end point of any interval $I_v \in \mathcal{I}$. Let v be the vertex such that $l_v = \text{pred}_L(l)$ and u be the vertex such that $l_u = \text{succ}_L(l)$.*

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 $l_w < l_{u'} \leq l_u \leq r_{u'} < r_w$, so w contains I_u . ◀

We will now assume that $G \cup \{w\}$ is a proper interval graph.

► **Lemma 9.** *$O(1)$ links need to be updated to transform $T_B(G)$ to $T_B(G \cup \{w\})$.*

Proof. By Lemma 7 for a vertex v , the parent in $T_B(G)$ is given by equation 2. By adding l_w and r_w , we may need to add an edge between w and $\text{parent}_{T_B(G \cup \{w\})}(w)$, and we also add links whenever w is the result of equation 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.

² If $l = l_v$, then one of w and v contain the other, and we can easily handle this case by aborting.

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 2 can be calculated using $\text{succ}_L(l_w)$ and $\text{succ}_R(l_v)$, then taking the minimum of the result.

To be complete, we will explicitly state the vertices that need to be relinked. If w is the new root of some component, then the old root is $l_r = \text{succ}_L(l_w)$ 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 points immediately precede l_w and r_w . Let u, v be the vertices such that $l_u = \text{pred}_L(l_w)$ and $l_v = \text{pred}_L(r_w)$. We may need to relink u and v as they may now be children of w . ◀

► **Lemma 10.** *The insert operation has time complexity $O(\lg n)$.*

Proof of Lemma 10. `insert` first checks that w is consistent with the other intervals in $O(\lg n)$ time. The transformation from $T_B(G)$ to $T_B(G \cup \{w\})$ 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`. Details for it and the following query, used in Section 4, are omitted. Given two vertices u, v (represented by intervals) not in G , compute $\text{dist}(u, v)$ in $G \cup \{u, v\}$ without requiring $G \cup \{u, v\}$ to be a proper interval graph. The main idea is that if $l_u < r_u < l_v$, then $\text{dist}(u, v)$ depends only on r_u and l_v . Thus we compute two vertices u', v' with $r_u = r_{u'}, l_v = l_{v'}$ so that $G \cup \{u', v'\}$ is a proper interval graph, and the distance can be computed by replacing u, v with u', v' . Thus we have:

► **Theorem 11.** *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`, `dist` in $O(\lg n)$ time, and `shortest_path` in $O(\lg n)$ time per vertex on the path. Furthermore, `dist` supports arguments not in the graph G : for intervals $x = [l_x, r_x], y = [l_y, r_y]$ not necessarily in G , $\text{dist}_{G \cup \{x, y\}}(x, y)$ is supported in $O(\lg n)$ time.*

4 Dynamic Interval Graphs in Incremental and Decremental Settings

In this section, we will study dynamic interval graphs in the incremental and decremental settings; Some details are omitted.

We do this by observing that any interval that is contained in some other interval can be removed without changing the length of the 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 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 is no longer contained by another interval (that is, becomes exposed), it will remain so until it is deleted. Hence, the total number of updates to the proper interval graph data structure will be $O(n)$. The two main theorems for this section are:

► **Theorem 12 (Incremental).** *There is a data structure that maintains an interval graph G in $O(n)$ words of space, where n is the total number of vertices to be inserted into G , and supports `dist` in $O(\lg n)$ worst-case time and `insert` in $O(\lg n)$ amortized time.*

► **Theorem 13** (Decremental). *There is a data structure that starts with a given n -vertex interval graph G and uses $O(n)$ words to support `dist` in $O(\lg n)$ worst-case time and `delete` in $O(\lg n)$ amortized time.*

As we briefly explained, these will be solved (in an amortized fashion) by reducing to the problem on dynamic proper interval graphs. Formally, for an interval graph G with intervals \mathcal{I} , we say that an interval $x \in \mathcal{I}$ is exposed if it is not contained by another interval of \mathcal{I} , and we let $\mathcal{I}^{\text{exposed}}(G)$ denote the set of all exposed interval of G . By the association between vertices and intervals, we will use the terms interval and vertex interchangeably. We note that by definition, the subgraph of G consisting of exposed vertices forms a proper interval graph. Let $x, y \in \mathcal{I}^{\text{exposed}}(G)$ be two exposed vertices, and by symmetry suppose that $l_x < l_y$. As x, y belong to a proper interval graph, we also have $r_x < r_y$. We will use $x < y$ to denote that $l_x < l_y$ for two exposed vertices, and $<$ defines a strict total order on the exposed vertices of the interval graph G (i.e. $<$ simply makes a comparison on the left endpoints of the given vertices, which must be unique). The following lemma allows us to reduce the distance query on interval graphs to the proper interval graph on the exposed vertices only.

► **Lemma 14.** *Given an interval graph G with fixed interval representation \mathcal{I} , its exposed intervals $\mathcal{I}^{\text{exposed}}(G)$ form a proper interval graph H with the following properties:*

- *Any interval x is contained by an interval of G iff x is contained by an interval of H .*
- *For any two vertices $x, y \in G$, $\text{dist}_G(x, y) = \text{dist}_{H \cup \{x, y\}}(x, y)$.*

Intuitively, we only need exposed intervals because, by definition of `parent` (equation 1), all `parent` intervals are exposed. Thus, Lemma 14 implies that, to support `dist` on interval graphs, it suffices to maintain an instance of fully dynamic proper interval graphs using Theorem 11, on the exposed vertices of G . Thus it remains to determine and maintain exposed vertices. To do so, in addition to using an instance of the fully dynamic proper interval graph structure, we will also store the exposed intervals in an auxiliary data structure:

► **Lemma 15.** *There exists a data structure using $O(n)$ words where n is the number of intervals current in the data structure, that can store the exposed intervals and support the following operations given an interval x in $O(\lg n)$ time: 1) determine whether x is contained by some interval currently in the data structure. 2) report all intervals that are contained by x and delete all of them, in $O(k \lg n)$ time, where k is the number of deleted intervals. 3) If x does not contain and is not contained by any interval currently in the data structure, insert it. 4) If x is in the data structure, return its predecessor or successor with respect to $<$, or report that it doesn't exist.*

We can use a red-black tree to store the exposed intervals using the left end points as the keys and the right end points as the values. This is sufficient to support the operations in Lemma 15. We now sketch our support for dynamic interval graphs under only insertions.

Proof Sketch of Theorem 12. We maintain the fully dynamic proper interval graph structure of Theorem 11 on the graph H whose vertices are the exposed intervals of G . We also maintain the binary search tree, T_H in Lemma 15 on the same intervals. Upon the insertion of an interval, if it is contained in some other interval, then we do nothing. If the new interval is exposed, then any interval it contains will no longer be exposed and must be deleted. As the deletion of intervals from the proper interval graph structure uses $O(\lg n)$ time and any exposed interval can only be deleted once, the total time over n insertions is $O(n \lg n)$. ◀

4.1 Data Structure for Decremental Interval Graphs

In this subsection, we consider the decremental case, where the updates are the deletion of intervals. First, we investigate how the set of exposed intervals change after a deletion.

► **Lemma 16.** *Let G be a proper interval graph, and $G' = G - x$ for some vertex $x \in G$. If x is not exposed in G , then, $\mathcal{I}^{\text{exposed}}(G) = \mathcal{I}^{\text{exposed}}(G')$.*

If x is exposed in G , then $\mathcal{I}^{\text{exposed}}(G) \setminus \{x\} \subseteq \mathcal{I}^{\text{exposed}}(G')$, and, for all $y \in \mathcal{I}^{\text{exposed}}(G') \setminus \mathcal{I}^{\text{exposed}}(G)$, $y \subseteq x$. Furthermore, $\mathcal{I}^{\text{exposed}}(G') = (\mathcal{I}^{\text{exposed}}(G) \setminus \{x\}) \dot{\cup} \{y \in \mathcal{I}^{\text{exposed}}(G') : x^- < y < x^+\}$ ($\dot{\cup}$ denotes a disjoint union), where x^- and x^+ are respectively the predecessor and the successor of x in $\mathcal{I}^{\text{exposed}}(G)$ (if either x^- or x^+ does not exist, then that constraint is omitted). That is, the newly added elements of $\mathcal{I}^{\text{exposed}}(G')$ are between x^- and x^+ .

In plain words, the lemma states that any new exposed intervals must fall between the predecessor and successor of the removed exposed interval.

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 these newly exposed intervals from the smallest to the largest.

► **Lemma 17.** *Consider a set of intervals S , and let $x \in S$ be an exposed interval. Let $x' \in S$ be the interval with minimum $l_{x'}$ (ties broken by largest $r_{x'}$) such that $r_{x'} > r_x$. Then, x' is exposed, and there does not exist any exposed interval $z \in S$ such that $x < z < x'$. If no such x' exists, then there is no exposed interval $w \in S$ such that $x < w$.*

Next we give a data structure which applies the criteria given in Lemma 17.

► **Lemma 18.** *There is an $O(n)$ -word data structure that can maintain a set of intervals (initially a given set, and not necessarily exposed) and support the following operations*

1. *Delete an interval in $O(\lg n)$ time;*
2. *Given any two exposed intervals x and y where $x < y$, report all exposed intervals z such that $x < z < y$ in $O((k+1) \lg n)$ time, where k is the number of returned intervals. We also allow $x = -\infty$ and/or $y = \infty$, in which case the constraint involving them is omitted.*

The structure is an augmented red-black tree storing all intervals, where the keys are the right endpoints of intervals (ties are broken by largest left end points of intervals), and to support the second operation, at each node, we store the minimum left endpoint of all the intervals in the subtree. Finally, we give a sketch of the data structure for supporting delete.

Proof sketch of Theorem 13. We build the data structures for the incremental case and also maintain T_G , the binary search tree in Lemma 18 on all the intervals of G . If the vertex x to be deleted is not exposed, then nothing needs to be done. If it is, then we find its predecessor and successor in $\mathcal{I}^{\text{exposed}}(G)$ using T_H , delete it, and find all the newly exposed vertices using Lemma 17 and T_G , and add them into the proper interval graph. As every interval can become exposed at most once, it is added into the proper interval graph at most once (and deleted at most once). Hence, over n deletes, the total run time is $O(n \lg n)$. ◀

5 Fully Dynamic Interval Graphs

The main goal of this section is to prove the following result:

► **Theorem 19.** *An interval graph G can be represented in $O(n)$ words to support `dist` in $O(n \lg n / S(n))$ time, `shortest_path` in $O(\lg n)$ time per vertex on the path, and `insert` and `delete` in $O(S(n) + \lg n)$ time, where n is the number of vertices currently in G and $S(n)$ is an arbitrary function that satisfies $S(n) = \Omega(1)$ and $S(n) = O(n)$. Setting $S(n) = \sqrt{n \lg n}$ yields an $O(n)$ -word solution with $O(\sqrt{n \lg n})$ -time support for `dist`, `insert` and `delete`.*

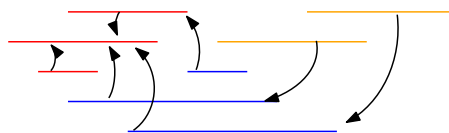
To prove this result, we first design a data structure to compute `parent` under updates. We decompose distance queries so that the structures can be more easily updated in Sections 5.1 and 5.2. Finally, we show how to update these in Section 5.3.

We define the `jump` of i as moving from the current vertex i to its `parent`(i) (as in Lemma 3)³. Then we can interpret Lemma 3 to a series of `jumps` from u to u_i , then to v . First, we give a data structure computing `jump`. By definition of `parent` (equation 1), the parent of a vertex i is the vertex $j = \arg \min_{r_j \geq l_i} l_j$. To compute this, we store all intervals in a red-black tree with their right endpoints as keys. At each node, we store both the minimal left endpoint among all intervals stored in the subtree rooted at that node and a pointer to the node containing that interval. We call this the *global interval tree*, and it uses $O(n)$ words. Given an interval i , to find j , we search for l_i to obtain a path (i.e. the nodes encountered on the standard binary search algorithm) and a set of subtrees (i.e. the subtrees rooted at right children of any node on the path, given that path continues towards the left child) containing the intervals v with $r_v \geq l_i$. For each we calculate the maximal left endpoint contained in the node (for those on the path) or the subtree and take the minimum. As the tree is balanced, this takes $O(\lg n)$ time. This tree can also be updated upon interval insertion or deletion in $O(\lg n)$ time. This simple structure is also a dynamic shortest path oracle for interval graphs.

5.1 Distance Computation

In this section, we will give an algorithm to compute the distance between two vertices x and y that is compatible with updates. The main idea is to break the interval graph into blocks of size $S(n)$. If we take the naive approach of calculating a shortest path, then the time complexity to compute the distance between two vertices x and y will depend on the distance itself. To combat this, we will decompose the path into subpaths each residing entirely within one of the blocks, and compute them quickly.

We sort the n given intervals by left endpoints, fix $S(n)$ and then divide the vertices (consecutively) into $\Theta(\frac{n}{S(n)})$ blocks, with each block containing $O(S(n))$ vertices (intervals). From left to right, we number blocks incrementally by B_1, B_2, \dots . For vertex i we say that the `jump` $i \rightarrow \text{parent}(i)$ is an *in-block jump* if both i and `parent`(i) belong to the same block, and it is an *out-of-block jump* otherwise.



■ **Figure 2** Depicting the parent relationship where the intervals are coloured to depict the decomposition into blocks of size 3.

³ Though `jump`(i) = `parent`(i), it flows more naturally when we describe it as an action, as it is a verb.

18:12 Distance Queries over Dynamic Interval Graphs

► **Example 20.** Consider figure 2. We see that any jump from the vertices represented by the red intervals are in-block jump, while jumps from all other vertices are out-of-block jumps. In particular, for any other vertex, since the jump is out-of-block, their parent in the distance tree of their block T_i (to be constructed in Section 5.2) would be different than their parent over all. However, for the red vertices these two parent relationships would coincide.

For any shortest path $x = p_1, \dots, p_k, y$ where $p_{i+1} = \text{parent}(p_i)$, we can decompose it into a sequence of in-block jumps followed by an out-of-block jump - and repeat. To compute the path length, we will compute the length of each of the in-block sequences and count the out-of-block jumps using

- **compressed-in-block-jump**(x , optional: y): Return the last vertex t such that all jumps between x, \dots, t are all in-block jumps, together with the distance between x and t . If y is given, also returns the distance between x and y or report that there is no path between them in the block.
- **jump**(x): given a vertex x , return its (global) parent.

Thus given two vertices x and y (with $l_y < l_x$ so that the block of x comes no earlier than the block of y) to the distance problem, we propose algorithm 1.

■ **Algorithm 1** Compressed computation of the distance between vertices x and y .

```

1:  $p, dist \leftarrow x, 0$ 
2: while  $p$  and  $y$  are not in the same block do
3:    $q, d(p, q) \leftarrow \text{compressed-in-block-jump}(p)$ 
4:   if  $q$  is adjacent to  $y$  then
5:     return  $dist + d(p, q) + 1$ 
6:    $p, dist \leftarrow \text{parent}(q), dist + d(p, q) + 1$ 
7:   if  $p = q$  (i.e.  $q = \text{parent}(q)$ ) then
8:     return unreachable
9:   if  $p$  is adjacent to  $y$  then
10:    return  $dist + 1$ 
11:  $q, d(p, q), d(p, y) \leftarrow \text{compressed-in-block-jump}(p, y)$ 
12: if  $l_y < l_q$  then
13:   if  $\text{parent}(q)$  is adjacent to  $y$  then
14:     return  $dist + d(p, q) + 1$ 
15:   else
16:     return unreachable
17: return  $dist + d(p, y)$ 

```

To show that this algorithm is correct, first we note that the blocks we visit is (weakly) monotonic. That is, at each jump, the block number never increases. Hence after each cycle of **compressed-in-block-jump** and **jump** the block number always decreases. This can be seen as by definition, for any x , $\text{parent}(x)$ has a smaller left endpoint, and as our blocks are obtained by sorting the left endpoints, it cannot be in a larger numbered block.

Proof. (Correctness of Algorithm 1) Let $s_1 > s_2 > s_3 \dots$ be the sequence of block numbers in our jump sequence. Let p_i be the first vertex in our path of block s_i and $q_i = \text{compressed-in-block-jump}(p_i)$ be the last vertex of block s_i . Let t be the block number of y . We have two cases: If $s_i > t > s_{i+1}$ for some i , then either q_i is adjacent to y or p_{i+1} is necessarily adjacent to y , the first we check on line 4, the second on line 9. The edge case here is if $s_i > t$ for all i , then at the end q_i would have no parent (which we defined

as $\text{parent}(q_i) = q_i$ as it is the end of its component, at which point we return that it is unreachable (on line 7-8). If $t = s_i$ for some i , then once we hit p_i , the while loop ends and we compute the remaining distance within the block. If q_i, y are in the same block with $l_{q_i} \leq l_y$, then the distance computation between p_i and y use only in-block jumps and thus is correct (line 17). Otherwise, if $l_{q_i} > l_y$, then we have the same situation as above. If it has a parent, then the parent must be adjacent to y (line 13-14). If it does not, then y cannot be reached (line 16). ◀

Given this, the time complexity of the distance algorithm is upper bounded by the time complexity of **compressed-in-block-jump** and **parent** multiplied by $O(\frac{n}{S(n)})$, as each operation is called at most $O(\frac{n}{S(n)})$ (the number of blocks) times.

5.2 Data Structures for Analyzing Jumps in Block

Now, we will discuss how to implement the **compressed-in-block-jump** subroutine after some preprocessing of each block. For each block B_i , we construct the distance tree, T_i (which may be a forest) of the interval graph G_i induced by the intervals in B_i . We precompute the depth, $\text{depth}(T_i, x)$, of each node x in T_i and also preprocess T_i using the approach of Bender and Farach-Colton [6] to provide constant-time support for the level ancestor operator, $\text{anc}(T_i, x, d)$, which, given a node x in T_i , returns the ancestor of x at depth d of T_i . The preprocessing time is $O(|B_i|)$, and T_i uses $O(|B_i|)$ words of space after preprocessing. By Lemma 3, depth and anc are sufficient to compute in constant time the distance, $\text{dist}(G_i, x, y)$, of two vertices x and y in G_i or determine that there is no path between them (i.e. when the two vertices belong to different trees in the forest). We will also store all the intervals of B_i using the left endpoint as keys in a red-black tree.

We propose the following **compressed-in-block-jump** algorithm (Algorithm 2).

■ **Algorithm 2** **compressed-in-block-jump**(x , optional: y).

```

1: if  $y$  exists then
2:   return  $\text{dist}(G_i, x, y)$  and also perform the below steps
3: Compute  $L_{\text{limit}}(i)$ 
4: Compute  $v_{\text{limit}}(i)$ 
5: if  $L_{\text{limit}}(i) \geq x$  then
6:   return  $x, 0$  as  $x$  jumps out of block already
7: Apply Lemma 3 to obtain  $z$  ( $u_i$  in Lemma 3) the last node before  $v_{\text{limit}}(i)$  on the path
   between  $x$  and  $v_{\text{limit}}(i)$  - or unreachable if a path cannot be found (i.e. they are in
   different subtrees).
8: if Line 7 is unreachable then
9:   return  $\text{anc}(T_i, x, 0), \text{depth}(x)$ 
10: if  $l_z \leq L_{\text{limit}}$  then
11:   return  $z, \text{depth}_{T_i}(x) - \text{depth}_{T_i}(z)$ 
12: else
13:   return  $\text{parent}_{T_i}(z), \text{depth}_{T_i}(x) - \text{depth}_{T_i}(z) + 1$ 

```

Define $L_{\text{limit}}(i) = \max\{r_w \mid l_w < \min\{l_v \mid v \in B_i\}\}$ and, if $L_{\text{limit}}(i) < \min\{l_v \mid v \in B_i\}$, we set it as $-\infty$ for convenience. This has the property that, for any $v \in B_i$ with $l_v < L_{\text{limit}}(i)$, $\text{jump}(v)$ is out-of-block as w (the interval achieving the maximum value in the definition of $L_{\text{limit}}(i)$) is a candidate for $\text{parent}(v)$ and is out-of-block. Conversely, $\text{jump}(v)$ for any

v with $l_v > L_{limit}(i)$ is in-block since any out-of-block jump from v would move $L_{limit}(i)$ to its right. Let $v_{limit}(i) = \arg \max_{\{v \in B_i | l_v < L_{limit}(i)\}} l_v$ be the last out-of-block jump vertex, computed by searching for $L_{limit}(i)$ over the left endpoints of the intervals in the block.

We may compute $L_{limit}(i)$ by descending the global interval tree. First search for the value $\min\{l_v | v \in B_i\}$ to obtain a path and a set of subtrees as right children of nodes on the path - these are all the intervals with right endpoint greater than $\min\{l_v | v \in B_i\}$. For each subtree and node on the path (in reverse in-order: starting with the subtree/node containing the interval with the largest right endpoint), we check if the minimal left endpoint in the subtree (or the left endpoint of the interval for a node on the path) is less than $\min\{l_v | v \in B_i\}$. If not, then we move on to the next subtree or node on the path. If so we return this interval for a node, and for a subtree we traverse it: for any node, if the right children's minimal left endpoint is less than $\min\{l_v | v \in B_i\}$, we move to it. Otherwise, we check the current node's interval and return it if its left endpoint is less than $\min\{l_v | v \in B_i\}$. If not, we move to the left child. If at the end, none of the subtrees nor nodes have a left endpoint smaller than $\min\{l_v | v \in B_i\}$, we return $-\infty$.

To find the first vertex with an out-of-block jump we compute $L_{limit}(i)$ and $v_{limit}(i)$. Then we apply $\text{dist}(v_{limit}(i), x)$ as in Lemma 3. Since $v_{limit}(i)$ is the boundary between the in-block jumps and out-of-block jumps, The penultimate vertex on the path to $v_{limit}(i)$ (i.e. u_i in Lemma 3) will also be on the boundary between being in-block and out-of-block (i.e. either it is the first out-of-block jump or it is the last in-block jump, and we check which by its relative order with $v_{limit}(i)$).

Correctness of compressed-in-block-jump. For computing the distance between x and y in the block, we call $\text{dist}(G_i, x, y)$ directly, which has been analyzed in Lemma 3 as a constant time operation with the constant-time support for depth and anc in T_i .

Furthermore, we want the number of jumps before we reach past $v_{limit}(i)$. If x and $v_{limit}(i)$ are in different trees in the forest T_i , then either x jumps out of block already, or we cannot reach $v_{limit}(i)$ via in-block jumps starting from x and the sequence of jumps ends at the root of the tree containing x . If they are in the same tree, we compute the path from x to $v_{limit}(i)$ and find the penultimate node on the path x_d (i.e. u_i in Lemma 3). If x_d jumps out of block, then x_{d-1} cannot, since otherwise it would also be adjacent to $v_{limit}(i)$. Similarly, if x_d does not jump out of block, then, as $\text{parent}(x_d) \leq v_{limit}(i)$ ($v_{limit}(i)$ is a candidate for the parent of x_d), $\text{parent}(x_d)$ must jump out of block. ◀

The query time of **compressed-in-block-jump** is $O(\lg n)$ since, all steps taken, such as the computation of $L_{limit}(i)$ and parent , are $O(\lg n)$ time. As the time complexity of the distance algorithm is the time complexity of **compressed-in-block-jump** and parent multiplied by $O(\frac{n}{S(n)})$, Algorithm 1 uses $O(n \lg n / S(n))$ time to answer a distance query.

5.3 Maintaining Data Structure under Update Operations

Update operations include vertex (interval) insertions and deletions. For any update, we have to maintain the following structures:

1. The global data structure of parent computation (i.e. the global interval tree).
2. The local parent structure of each block (i.e T_i for each block B_i).
3. The block structure of the whole graph.

As discussed the global interval tree can be maintained in $O(\lg n)$ time for each update. This part is independent of the block structure. As the structure for each local block only uses the intervals of that block, for each update, we need only update the structures for the

block containing the inserted or deleted interval. Since the update will change the interval set of this block, we rebuild the local structures in $O(S(n))$ time. Thus overall each update has complexity $O(S(n) + \lg n)$.

Note that we assume the block size stays $O(S(n))$ for the previous analysis. However, the block size will change whenever an update occurs in it. In the worst case, all updates would occur in the same block, and thus we must be able to maintain our block sizes to be $\Theta(S(n))$. Moreover, the total number of intervals n will also change, and thus $S(n)$ will also change. The efficiency may not be guaranteed if the total number of intervals no longer correlates with our block size. Therefore, we use two processes, **Split** and **Rebuild**, to maintain block sizes of $\Theta(S(n))$. In general, **Split** happens whenever a block's size is too large, which might degenerate the complexity of **compressed-in-block-jump**. A **Rebuild** will occur after a certain number of updates in order to control the number of blocks and ensure that the block size corresponds to the current $S(n)$ value.

Split: The key part of the analysis is that we assume the block size is $O(S(n))$ all the time. The **Split** process is for maintaining this property. After any **insert**, if the size of the block containing the new vertex is $2S(n)$, we will split the block into two blocks, and each block has $S(n)$ intervals sorted by left endpoints. As we need to rebuild two blocks of size $S(n)$, the time needed is $O(S(n))$. Finally, we note that since every block begins with $S(n)$ vertices and has $S(n)$ vertices after a split, at least $S(n)$ inserts must occur in a block to split it.

Rebuild: This process denotes a complete rebuilding of the whole block structures, including block dividing and all local preprocessing of blocks. Since the cost of building the structure for each block is linear, the total cost is $O(n)$. The motivation of this operation comes from two causes. Firstly, when the number of blocks increases greatly, the compressed distance computing, which is dominated by the number of blocks, will degenerate. Secondly, denoting the number of intervals in the last **Rebuild** as n' , if n' differs greatly from n , $S(n)$ may also be different enough from $S(n')$, so that our complexity will not be $O(n \lg n / S(n))$ per query or $O(S(n) + \lg n)$ per update (corresponding to the current $S(n)$). Therefore, we trigger a **Rebuild** after every $\frac{n'}{2}$ updates. After all $\frac{n'}{2}$ updates, the current number of intervals n stays within $[\frac{n'}{2}, \frac{3n'}{2}]$, and at most $\frac{n'}{2S(n')}$ blocks are created or destroyed. To see this, to destroy a block requires $S(n)$ deletions and to add a block also needs $S(n)$ insertions to trigger a **Split**. Thus the number of blocks remains $\Theta(\frac{n'}{S(n')}) = \Theta(\frac{n}{S(n)})$. Since $S(n)$ is a function that satisfies $S(n) = O(n)$, the complexity of all these n' updates and queries in between stays the same corresponding to the previous $S(n')$.

To deamortize, whenever we **Rebuild**, we create a new structure over the next $\frac{n}{4} = \Theta(n')$ updates containing the contents of the original structure (using the new $S(n)$ as our block size) and the $\frac{n}{4}$ updates. While we rebuild, we also perform the updates in the old structure and answer queries using the old structure. Thus for each of the next $\frac{n}{4}$ updates, we incur an extra $O(S(n'))$ time per update. Upon the completion of the rebuild, we switch to using the newly created structure. Lastly, we summarize every part and prove the main theorem here.

Proof of Theorem 19. For an interval graph G that the number of vertices is currently n , in our algorithm, we only maintain a global structure to maintain **parents** and do local preprocessing for answering **depth** and **anc** in each block. The **parent** data structure takes $O(n)$ space. The preprocessing of each of the $\Theta(\frac{n}{S(n)})$ blocks occupies $O(T)$ words (T denotes the number of intervals in this block). The aggregation gives $O(n)$ space in total. Since $S(n) = O(n)$, the interval graph is therefore represented in $O(n)$ words of space. Distance

queries takes $O(n \lg n / S(n))$ time. For any update, we showed in Section 5.3, that it takes $O(S(n) + \lg n)$ time. Moreover, the extra `Split` and `Rebuild` processes only guarantee the cost is not degenerated and does not affect the complexity per operation. Hence, we have our proof for the main theorem. ◀

6 A Lower Bound for Axis-Aligned Line Segments in 3D

In this section, we show that the problem of supporting distance queries over dynamic intersection graphs of 3D axis-aligned line segments is conditionally hard by reducing from the online matrix-vector multiplication problem.

► **Definition 21** (Online Boolean Matrix Vector Multiplication (OMv)). *Let M be a $n \times n$ boolean matrix, and let v_1, \dots, v_n be a set of $n \times 1$ vectors. We must compute and output Mv_i for each i before receiving the next vector.*

The corresponding hardness conjecture is by Henzinger et al. [19]. For any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$ -time algorithm that solves OMv with error probability at most $1/3$.

► **Theorem 22.** *If updates and distance queries over an intersection graph of 3D axis-aligned line segments can be respectively supported in $O(Q(n))$ and $O(T(n))$ time, then OMv can be solved in $O(n^2(T(n) + Q(n)))$ time. Thus, for any constant $\varepsilon > 0$, $T(n)$ and $Q(n)$ can not both be $O(n^{1-\varepsilon})$, unless the OMv conjecture is false. Here n is the length of the vectors. If \hat{n} is the number of vertices of the graph, then $T(\hat{n})$ and $Q(\hat{n})$ cannot both be $O(\hat{n}^{1/2-\varepsilon})$.*

Proof. For each $i \in [1, n]$, define a line segment X_i between end points $(0, i, 0)$ and $(2n, i, 0)$, and a segment Y_i between $(i, 0, 1)$ and $(i, 2n, 1)$. For each entry $M_{i,j} = 1$, we create a line segment $Z_{j,i}$ between $(j, i, 0)$ and $(j, i, 1)$. We will refer to the 3 types of line segments as type X , type Y and type Z segments. Furthermore, we add the line segment X_{n+1} whose end points are $(0, n+1, 0)$ and $(2n, n+1, 0)$ to represent the incoming vector. When given a vector v_i , we add the following type Z segments. For each entry $v_i(j) = 1$, we add a type Z segment $Z_{j,n+1}$ with end points $(j, n+1, 0)$ and $(j, n+1, 1)$. By construction, both type X and type Y segments are only adjacent to type Z segments, and each type Z segment $Z_{i,j}$ is only adjacent to two other segments X_j and Y_i .

We now claim that $Mv_i(j) = 1$ if and only if $\text{dist}(X_j, X_{n+1}) = 4$. First suppose that $\text{dist}(X_j, X_{n+1}) = 4$. Then by construction, there exists an index w such that the path is $X_j, Z_{w,j}, Y_w, Z_{w,n+1}, X_{n+1}$. This implies that $M(j, w) = 1$ and $v_i(w) = 1$, and thus $Mv_i(j) = 1$. Conversely, suppose that $Mv_i(j) = 1$. Then there exists an index w such that $M(j, w) = v_i(w) = 1$. Thus $X_j, Z_{w,j}, Y_w, Z_{w,n+1}, X_{n+1}$ is a path of length 4, so $\text{dist}(X_j, X_{n+1}) \leq 4$. To see that it cannot be strictly less than 4, we note that, since each type Z segment is adjacent to a single type X segment and a single type Y segment, we may view it as a subdivision of an edge between its two incident segments. Since segments of types X (and Y) are never adjacent to segments of the same type, if we contract vertices representing type Z segments, we are left with a bipartite graph. If $\text{dist}_G(X_j, X_{n+1}) < 4$, it must be 2 (by the subdivision of edges), but it cannot be 2 as that implies two type X segments are adjacent.

Thus, the computation of a single Mv_i operation is reduced to $O(n)$ insertions and deletions of segments of type Z , and n distance queries. Over all n operations, this incurs $O(n^2)$ updates and $O(n^2)$ queries. Therefore, $O(n^2T(n) + n^2Q(n))$ cannot be $O(n^{3-\varepsilon})$ for any constant $\varepsilon > 0$ unless the OMv conjecture is false. Finally, we note that by construction, the number of vertices in the graph is at most $\hat{n} = O(n^2)$ and the theorem follows. ◀

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. doi:10.1007/978-3-642-24100-0_39.
- 2 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct encodings for families of interval graphs. *Algorithmica*, 83(3):776–794, 2021. doi:10.1007/s00453-020-00710-w.
- 3 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. doi:10.1109/FOCS46700.2020.00057.
- 4 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees. *ACM Transactions on Algorithms*, 1, December 2003. doi:10.1145/1103963.1103966.
- 5 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001. doi:10.1145/502102.502107.
- 6 Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.
- 7 Karl Bringmann, Sándor Kisfaludi-Bak, Marvin Künnemann, André Nusser, and Zahra Parsaeian. Towards sub-quadratic diameter computation in geometric intersection graphs. In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7–10, 2022, Berlin, Germany*, volume 224 of *LIPICs*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.SoCG.2022.21.
- 8 Timothy M. Chan. Finding triangles and other small subgraphs in geometric intersection graphs. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22–25, 2023*, pages 1777–1805. SIAM, 2023. doi:10.1137/1.9781611977554.ch68.
- 9 Timothy M. Chan and Dimitrios Skrepetos. All-pairs shortest paths in geometric intersection graphs. *J. Comput. Geom.*, 10(1):27–41, 2019. doi:10.20382/jocg.v10i1a2.
- 10 Timothy M. Chan and Dimitrios Skrepetos. Approximate shortest paths and distance oracles in weighted unit-disk graphs. *J. Comput. Geom.*, 10(2):3–20, 2019. doi:10.20382/jocg.v10i2a2.
- 11 Danny Z. Chen, D. T. Lee, R. Sridhar, and Chandra N. Sekharan. Solving the all-pair shortest path query problem on interval and circular-arc graphs. *Networks*, 31(4):249–258, 1998. doi:10.1002/(SICI)1097-0037(199807)31:4<249::AID-NET5>3.0.CO;2-D.
- 12 Jonathan B. Conroy and Csaba D. Tóth. Hop-spanners for geometric intersection graphs. In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry, SoCG 2022, June 7–10, 2022, Berlin, Germany*, volume 224 of *LIPICs*, pages 30:1–30:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.SoCG.2022.30.
- 13 Christophe Crespelle. Fully dynamic representations of interval graphs. *Theoretical Computer Science*, 759:14–49, 2019. doi:10.1016/j.tcs.2019.01.007.
- 14 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004. doi:10.1145/1039488.1039492.
- 15 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.
- 16 Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM J. Comput.*, 35(1):151–169, 2005. doi:10.1137/S0097539703436357.
- 17 Cyril Gavoille and Christophe Paul. Optimal distance labeling for interval graphs and related graph families. *SIAM J. Discret. Math.*, 22(3):1239–1258, 2008. doi:10.1137/050635006.

- 18 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. doi:10.4230/LIPICs.ISAAC.2020.25.
- 19 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, STOC '15, pages 21–30, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2746539.2746609.
- 20 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. doi:10.1109/FOCS52979.2021.00044.
- 21 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. doi:10.1137/1.9781611976465.149.
- 22 J. Ian Munro and Corwin Sinnamon. Time and space efficient representations of distributive lattices. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 550–567. SIAM, 2018. doi:10.1137/1.9781611975031.36.
- 23 J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao, editors, *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, volume 123 of *LIPICs*, pages 67:1–67:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
- 24 Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.
- 25 Gaurav Singh, N. S. Narayanaswamy, and G. Ramakrishna. Approximate distance oracle in $o(n^2)$ time and $o(n)$ space for chordal graphs. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation – 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, volume 8973 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2015. doi:10.1007/978-3-319-15612-5_9.
- 26 Mikkel Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory – SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2004. doi:10.1007/978-3-540-27810-8_33.
- 27 Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. Searching the wikipedia with contextual information. In James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz, Key-Sun Choi, and Abdur Chowdhury, editors, *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 1351–1352. ACM, 2008. doi:10.1145/1458082.1458274.
- 28 Monique V. Vieira, Bruno M. Fonseca, Rodrigo Damazio, Paulo Braz Golgher, Davi de Castro Reis, and Berthier A. Ribeiro-Neto. Efficient search ranking in social networks. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 563–572. ACM, 2007. doi:10.1145/1321440.1321520.

- 29 Harry Wiener. Structural determination of paraffin boiling points. *J. Am. Chem. Soc.*, 69(1):17–20, 1947.
- 30 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. *Comput. Appl. Biosci.*, 10(3):309–317, 1994. doi:10.1093/bioinformatics/10.3.309.