A Simple Polynomial Time Algorithm for Max Cut on Laminar Geometric Intersection Graphs

Utkarsh Joshi
Department of Computer Science and Automation, Indian Institute of Science, Bengaluru, India

Saladi Rahul
Department of Computer Science and Automation, Indian Institute of Science, Bengaluru, India

Josson Joe Thoppil
Department of Information Technology, National Institute of Technology, Karnataka, India

Abstract

In a geometric intersection graph, given a collection of n geometric objects as input, each object corresponds to a vertex and there is an edge between two vertices if and only if the corresponding objects intersect. In this work, we present a somewhat surprising result: a polynomial time algorithm for max cut on *laminar* geometric intersection graphs. In a laminar geometric intersection graph, if two objects intersect, then one of them will completely lie inside the other. To the best of our knowledge, for max cut this is the first class of (non-trivial) geometric intersection graphs with an exact solution in polynomial time. Our algorithm uses a simple greedy strategy. However, proving its correctness requires non-trivial ideas.

Next, we design almost-linear time algorithms (in terms of n) for laminar axis-aligned boxes by combining the properties of laminar objects with vertical ray shooting data structures. Note that the edge-set of the graph is *not* explicitly given as input; only the n geometric objects are given as input.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Graph algorithms analysis

Keywords and phrases Geometric intersection graphs, Max cut, Vertical ray shooting

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2022.21

1 Introduction

In the *maximum cut* (a.k.a., max cut) problem, the input is an undirected graph, and the goal is to partition the vertex set into two disjoint sets such that the number of edges having their endpoints in different sets is maximized. Max cut is a classical optimization problem. Weighted version of it is in fact one of Karp's original 21 NP-hard problems [24]. Max cut has been studied extensively on several classes of the graphs. It remains NP-hard even for the special classes of the graphs like cubic graphs [5], total graphs [18], chordal graphs, split graphs, co-bipartite graphs, tripartite graphs [7], and permutation graphs (recently shown [14]). As a result, designing approximation algorithms for max cut has naturally received a lot of attention [13, 17, 26, 23]. On the other hand, the classes of graphs which are of relevance in the context of our work are those for which max cut has been shown to be polynomial time solvable such as planar graphs [19], line graphs [18], graphs with bounded treewidth [7], co-bipartite chain graphs [10], graphs not contractible to K_5 [3] and split-indifference graphs [6]. In this work we add another class of graphs to this list.

Geometric intersection graphs

Consider a set $\mathcal{O} = \{O_1, \ldots, O_n\}$ which is a collection of n geometric objects (such as intervals and rectangles). In a *geometric intersection graph*, each object corresponds to a vertex and there is an edge between two vertices if and only if the corresponding objects intersect. We

restate the max cut problem in the context of geometric intersection graphs: Color each object in $\mathcal{O}$ either red or green so that the maximum number of pairs in $\mathcal{O}$ *contribute* to the cut. A pair (O_i, O_j) contributes to the cut iff (a) O_i and O_j intersect, and (b) O_i and O_j have different colors.

Max cut on geometric intersection graphs

The complexity of the max cut on interval graphs was unknown for a long time and was mentioned as an open problem way back in 1985 in celebrated paper of Johnson [22]. At SoCG'21, Adhikary et al. [1] showed that max cut is NP-complete *even* for interval graphs. Unit interval graphs are a special case of interval graphs where each interval has unit length. Surprisingly, the complexity status of the max cut on the class of unit interval graphs is still unknown. Two previous results [8, 9] claimed a polynomial time algorithm for unit interval graphs, but they were reported as incorrect later [6, 25]. Recently, in a series of improvements, it was first shown that max cut is NP-complete on interval graphs of interval count four [15] and then for interval count two [4], which is a special case of interval graphs but more general than unit-interval graphs. On the approximation side, a PTAS can be obtained for max cut on unit-interval graphs ([21] shows a PTAS for the *max bisection* problem, but their technique can be adapted for the max cut problem as well). In 2D, max cut on unit-disk graphs [16] has been shown to be NP-hard.

Exact max cut on laminar geometric intersection graphs

The starting point of our work is the following set of observations. Any two general intervals can have two types of intersections: either (a) partially overlap, or (b) one interval lies inside the other. Unit-interval graphs capture the first type of intersection. From the discussion above, it is clear that max cut on unit-interval graphs has received a lot of interest in terms of designing an exact algorithm, or proving its hardness, or the PTAS result. On the other hand, *laminar interval graphs* can capture the second type of intersection, i.e., if two intervals intersect then one of them lies completely inside the other. However, to the best of our knowledge, there has not been any prior work on max cut for laminar interval graphs.

Motivated by this, we study max cut on *laminar geometric intersection graphs*. In a laminar geometric intersection graph, if two objects intersect, then one of the object will lie completely inside the other object (see Figure 1(a) for an example). Somewhat surprisingly, we prove that a polynomial time *exact* algorithm exists for laminar geometric intersection graphs. To the best of our knowledge, for max cut this is the first class of (non-trivial) geometric intersection graphs with an exact polynomial time solution. We propose a simple greedy algorithm to compute the max cut on laminar geometric intersection graphs. However, as is the case with many greedy algorithms, our proof of optimality involves several non-trivial arguments.

Designing a fast algorithm

The following discussion assumes a basic familiarity with computational geometry. Note that the edge-set of the graph is *not* explicitly given as input; only the n geometric objects are given as input. As a result, a fruitful line of research is to exploit the small input size and the geometry of the intersection graph to design almost-linear time (in terms of n) algorithms to compute max cut. As concrete applications, in this work we consider laminar axis-aligned boxes intersection graphs in $3D$ and $2D$. The first step of our algorithm is to compute a *tree representation* of the laminar objects (see Figure 1 for an example). Using range searching

data structures [2] in a standard manner one can construct such a tree representation in $O(n \log^c n)$ time for axis-aligned boxes in $3D$ for a sufficiently large constant c. Instead, we perform a sweep-plane and use the properties of laminar objects to reduce the original problem to *dynamic 2D vertical ray shooting problem*. As a result, we are able to compute the tree representation in $O\left(n \frac{\log n}{\log \log n}\right)$ expected time for axis-aligned boxes in $3D$. Adapting our algorithm for axis-aligned rectangles in $2D$ leads to an $O(n \log \log U)$ expected time algorithm, where the endpoints of the boxes lie on the integer grid $[U]^2$. The model of computation is the word-RAM model.

2 Laminar geometric intersection graphs

In this section, we will present an exact solution for max cut on laminar geometric intersection graphs. In a laminar geometric intersection graph, if two objects intersect, then one of the objects will lie completely inside the other object (see Figure 1(a) for an example).

2.1 Tree representation and our algorithm

A laminar geometric intersection graph can naturally be represented as a tree. Let $\mathcal{O} = \{O_1, \ldots, O_n\}$ be a collection of n laminar geometric objects. An object O_i is said to *dominate* object O_j if and only if O_j lies completely inside O_i. Define level-1 objects of $\mathcal{O}$ to be those objects which are not dominated by anyone in $\mathcal{O}$. Call them $\mathcal{O}_1$. In Figure 1, $\mathcal{O}_1 = \{O_1, O_2\}$. For an integer $i \geq 2$, level-i objects are obtained by computing the set of objects in $\mathcal{O} \setminus \bigcup_{j=1}^{i-1} \mathcal{O}_j$ which are not dominated by any other object in $\mathcal{O} \setminus \bigcup_{j=1}^{i-1} \mathcal{O}_j$. Call them $\mathcal{O}_i$. In Figure 1, $\mathcal{O}_2 = \{O_3, O_4, O_5, O_6, O_7\}$. We stop the recursion when $\mathcal{O} \setminus \bigcup_{j=1}^{\ell-1} \mathcal{O}_j$ becomes empty, for some integer ℓ.

Now we are ready to describe the tree representation. We will initialise $|\mathcal{O}_1|$ trees, and each object in $\mathcal{O}_1$ will be made the root of one of the tree (see Figure 1(b)). For $i \in [2, \ell-1]$, each object $O \in \mathcal{O}_i$ will be made the child of that node in level-$(i-1)$ whose corresponding object contains O. See Figure 1. Note that max cut can be computed independently for each tree and then trivially combined to obtain the final solution. As such, without loss of generality we can compute the max cut for a single tree. The following property is easy to observe.

▶ **Lemma 1.** *Denote the tree representation of $\mathcal{O}$ by $\mathcal{T}$. Two objects O and O' intersect if and only if O is an ancestor of O' or O' is an ancestor of O in $\mathcal{T}$.*

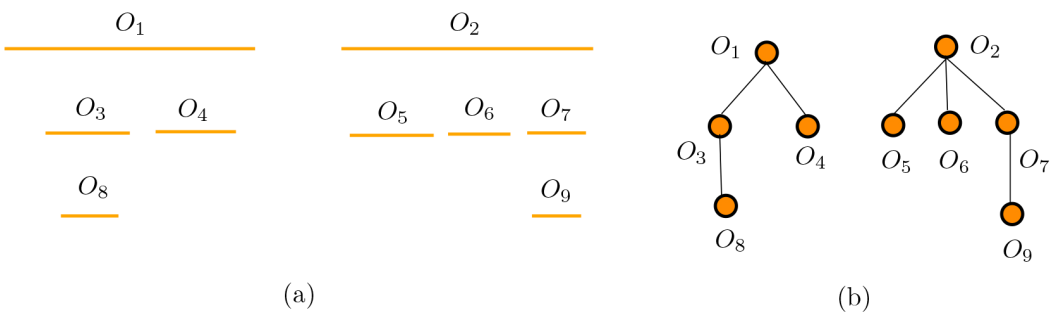

Figure 1 (a) An input instance of nine laminar intervals on the real-line. For the sake of clarity, the intervals have been drawn vertically apart, (b) The corresponding tree representation.

From now on we will work with the tree representation $\mathcal{T}$ of $\mathcal{O}$. Each node in $\mathcal{T}$ will be colored either red or green. In this new terminology, given two nodes $v_i \in \mathcal{T}$ and $v_j \in \mathcal{T}$, the pair (v_i, v_j) *contributes* to the cut iff (a) v_i is an ancestor of v_j, and (b) v_i and v_j have different colors. The goal is to maximize the number of pairs in $\mathcal{T}$ which contribute to the cut.

For any node $u \in \mathcal{T}$, let $A(u)$ be the ancestors of u (excluding u) in $\mathcal{T}$ and let $D(u)$ be the nodes in the subtree of u (excluding u).

The algorithm can now be described in a single line:

Algorithm 1 Laminar geometric intersection graphs.

– For each node $u \in \mathcal{T}$, if $|D(u)| \geq |A(u)|$, then color u green; otherwise, color u red.

See Figure 2 for an example. Proving that Algorithm 1 actually reports the maximum cut turns out to be a non-trivial exercise. Our proof will have two steps. First, in Lemma 2 we prove that among all colorings which do not create *inversions* in $\mathcal{T}$, none of them lead to a larger cut value than the cut value returned by Algorithm 1. A coloring of the nodes in $\mathcal{T}$ is said to create an inversion if there is at least one node which is colored green and its parent node is colored red. Next, in Lemma 3 we prove that among *all* possible ways of coloring $\mathcal{T}$, it suffices to consider colorings which do not create *inversions* in $\mathcal{T}$. This proves that Algorithm 1 indeed computes the max cut.

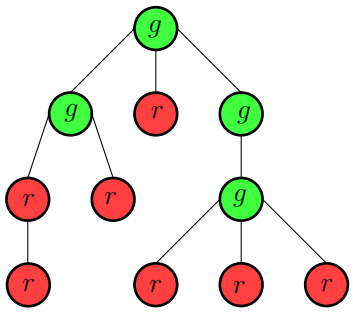

Figure 2 Coloring produced by Algorithm 1 on a tree with eleven nodes. Green colored nodes are labelled g and the red colored nodes are labelled r.

2.2 Optimality of our algorithm

▶ **Lemma 2.** *For any coloring $\mathcal{C}$, let $Cut(\mathcal{C})$ be the number of pairs in $\mathcal{T}$ contributing to the cut. Let $\mathcal{C}_{ni}$ be any coloring of $\mathcal{T}$ which creates no inversions, and let $\mathcal{C}_{alg}$ be the coloring of $\mathcal{T}$ by our algorithm. Then $Cut(\mathcal{C}_{ni}) \leq Cut(\mathcal{C}_{alg})$.*

Proof. We will start with coloring $\mathcal{C}_{\text{ni}}$ of the tree and perform n iterations. For $1 \leq i \leq n$, in the i-th iteration, we might potentially *flip* the color of one of the nodes in the tree. Let $\mathcal{C}_i$ be the coloring of the tree at the end of the i-th iteration, for all $1 \leq i \leq n$, and let $\mathcal{C}_0 = \mathcal{C}_{\text{ni}}$. We will maintain the following three invariants:

1. $Cut(\mathcal{C}_{i-1}) \leq Cut(\mathcal{C}_i)$, for all $1 \leq i \leq n$, i.e., the cut value does not decrease after each iteration.
2. $\mathcal{C}_n = \mathcal{C}_{\text{alg}}$, i.e., at the end of the n iterations, the (potentially) new coloring in the tree will correspond to $\mathcal{C}_{\text{alg}}$.
3. $\mathcal{C}_i$ induces no inversion in $\mathcal{T}$, for all $1 \leq i \leq n$.

From the first two invariants, it follows that $Cut(\mathcal{C}_{\text{ni}}) = Cut(\mathcal{C}_0) \leq Cut(\mathcal{C}_1) \leq Cut(\mathcal{C}_2) \leq \ldots \leq Cut(\mathcal{C}_n) = Cut(\mathcal{C}_{\text{alg}})$. Now we present the details of the reduction from coloring $\mathcal{C}_{\text{ni}}$ to $\mathcal{C}_{\text{alg}}$.

With respect to the coloring $\mathcal{C}_{\text{alg}}$ of the tree $\mathcal{T}$, let $\mathcal{G}$ be the set of nodes in $\mathcal{T}$ colored green. Since $\mathcal{C}_{\text{alg}}$ induces no inversion in $\mathcal{T}$, nodes in $\mathcal{G}$ form a tree (and not a forest). Lets call this tree $\mathcal{T}_{\mathcal{G}}$. The remaining portion of $\mathcal{T}$, i.e., $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$ will be a collection of trees (a.k.a. a forest) whose nodes are colored red. Unlike the traditional way of either a top-down or a bottom-up traversal of a tree, we will perform a more careful traversal of $\mathcal{T}$: First, traverse the nodes in $\mathcal{T}_{\mathcal{G}}$ in a top-down manner. Next, traverse the trees in $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$ in a bottom-up manner. See Figure 3. Each iteration corresponds to visiting one of the n nodes in $\mathcal{T}$.

Top-down traversal of $\mathcal{T}_{\mathcal{G}}$. We will start from the coloring induced by $\mathcal{C}_{\text{ni}}$ on tree $\mathcal{T}_{\mathcal{G}}$. We will perform a top-down traversal of $\mathcal{T}_{\mathcal{G}}$ and at the end of the $|\mathcal{G}|$ iterations (i.e., visiting $|\mathcal{G}|$ nodes), all the nodes in $\mathcal{T}_{\mathcal{G}}$ will be colored green.

Let $u \in \mathcal{T}_{\mathcal{G}}$ be the node visited in the i-th iteration. If u is colored green, then no flip operation will be performed, and we will set $\mathcal{C}_i = \mathcal{C}_{i-1}$. The interesting case is when u is colored red, where we will flip the color of u to green. Since $\mathcal{C}_{i-1}$ is a non inversion coloring, all the $|D(u)|$ nodes of $\mathcal{T}$ in the subtree of u will be colored red. Also, due to the top-down traversal of $\mathcal{T}_{\mathcal{G}}$, all the $|A(u)|$ ancestors of u will be colored green. Since Algorithm 1 colored u green, it must be the case that $|D(u)| \geq |A(u)|$. Therefore, flipping the color of u from red to green will ensure that $Cut(\mathcal{C}_i) = Cut(\mathcal{C}_{i-1}) - |A(u)| + |D(u)| \geq Cut(\mathcal{C}_{i-1})$ (satisfying Invariant 1). Also, after the flip operation, the coloring induced by $\mathcal{C}_i$ will have no inversion (satisfying Invariant 3).

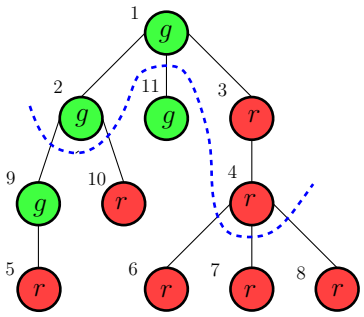

Figure 3 Arbitrary C_{ni} for a tree with eleven nodes. Green colored nodes are labelled g and the red colored nodes are labelled r. The nodes indexed 1 to 4 belong to $\mathcal{T}_{\mathcal{G}}$ and the nodes indexed 5 to 11 belong to $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$. Indexing denotes the order in which the nodes will be traversed.

Bottom-up traversal of $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$. At the end of performing $|\mathcal{G}|$ iterations on $\mathcal{T}_{\mathcal{G}}$, we will have the coloring induced by $\mathcal{C}_{|\mathcal{G}|}$. Now consider any tree in $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$. We will perform a bottom-up traversal of that tree. Let u be a node visited in the i-th iteration. If u is colored red, then no flip operation will be performed and we will set $\mathcal{C}_i = \mathcal{C}_{i-1}$. The interesting case is when u is colored green, where we will flip the color of u to red. Since $\mathcal{C}_{i-1}$ is a no inversion coloring, all the $|A(u)|$ ancestors of u in $\mathcal{T}$ will be colored green. Also, due to bottom-up traversal of the tree, all the $|D(u)|$ nodes in the subtree of u in $\mathcal{T}$ will be colored red. Since Algorithm 1 colored u red, it must be the case that $|D(u)| < |A(u)|$. Therefore, flipping the color of u from green to red will ensure that $Cut(\mathcal{C}_i) = Cut(\mathcal{C}_{i-1}) - |D(u)| + |A(u)| \geq Cut(\mathcal{C}_{i-1})$ (satisfying Invariant 1). Also, after the flip operation, the coloring induced by $\mathcal{C}_i$ will create no inversions (satisfying Invariant 3). Perform the above operations for all the trees in $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$.

At the end of the n iterations, it is easy to see that the coloring in $\mathcal{T}$ corresponds to $\mathcal{C}_{alg}$: the top-down traversal ensures that all the nodes in $\mathcal{G}$ are colored green and all the nodes in $\mathcal{T} \setminus \mathcal{T}_{\mathcal{G}}$ are colored red, and hence, satisfying Invariant 2. ◀

2.3 A non-inversion coloring of the tree

In this subsection, we will prove the following result.

▶ **Lemma 3.** *Let $\mathcal{C}$ be an arbitrary coloring of the tree $\mathcal{T}$ and let $Cut(\mathcal{C})$ be the number of pairs in $\mathcal{T}$ contributing to the cut. Then there exists a coloring $\mathcal{C}_{ni}$ which creates no inversions in $\mathcal{T}$ and $Cut(\mathcal{C}_{ni}) \geq Cut(\mathcal{C})$.*

Fix an ordering from left-to-right of the m leaves in $\mathcal{T}$. The ordering of the leaves naturally fixes an order of all the nodes at each level of $\mathcal{T}$. For all $1 \leq i \leq m$, let π_i be the root-to-leaf path for the i-th leaf of $\mathcal{T}$. We start with coloring $\mathcal{C}$ of the tree $\mathcal{T}$, and let $\mathcal{C}_0 = \mathcal{C}$. We will perform m *iterations* and in each iteration potentially flip the color of some of the nodes in $\mathcal{T}$. For all $1 \leq i \leq m$, let $\mathcal{C}_i$ be the coloring of the tree at the end of the i-th iteration. At the end of the i-th iteration, for all $1 \leq i \leq m$, we will ensure that the following invariants are maintained:

1. For all $1 \leq j \leq i$, there are no inversions among nodes on path π_j.
2. $Cut(\mathcal{C}_i) \geq Cut(\mathcal{C}_{i-1})$.

By the first invariant it is guaranteed that the coloring induced by $\mathcal{C}_m$ creates no inversions in $\mathcal{T}$. Therefore, we set $\mathcal{C}_{\text{ni}} = \mathcal{C}_m$. Via the second invariant, it follows that $Cut(\mathcal{C}_{\text{ni}}) \geq Cut(\mathcal{C})$. This will prove Lemma 3. For any $1 \leq i \leq m$, we now present the details of the transformation from $\mathcal{C}_{i-1}$ to $\mathcal{C}_i$. First, we will perform a *modification* step.

Modification steps. At the beginning of the i-th iteration, let $g(\pi_i)$ and $r(\pi_i)$ be the number of green and red colored nodes on path π_i, respectively. We will *modify* the color of the nodes in π_i as follows:

- If $g(\pi_i) \geq r(\pi_i)$, then the first $g(\pi_i)$ nodes on path π_i are colored green, and the remaining $r(\pi_i)$ nodes are colored red.
- Otherwise, the first $r(\pi_i)$ nodes on path π_i are colored green, and the remaining $g(\pi_i)$ nodes are colored red.

Let $E = \{(u, v) | u, v \in \mathcal{T}$ and u is an ancestor of $v\}$. For any node $u \in \mathcal{T}$, let $u_1, u_2, \ldots, u_t$ be its children in the left-to-right ordering. Then we define $\textsc{left}(u, u_j) = \{u_1, u_2, \ldots, u_{j-1}\}$ and $\textsc{right}(u, u_j) = \{u_{j+1}, \ldots, u_t\}$. Also, let $\pi_i(1), \pi_i(2), \ldots$ be the sequence of nodes on π_i from root-to-leaf. Now we define two sets $\mathcal{R}_i$ and $\mathcal{L}_i$ as follows:

$$\mathcal{R}_i = \{v | v \in \textsc{right}(\pi_i(j), \pi_i(j+1)) \text{ and } \pi_i(j) \in \pi_i\} \text{ and}$$
$$\mathcal{L}_i = \{v | v \in \textsc{left}(\pi_i(j), \pi_i(j+1)) \text{ and } \pi_i(j) \in \pi_i\}$$

See Figure 4 for an example. Now we will partition E into four subsets $E_1, E_2, E_3,$ and E_4. We will argue that the contribution of $E_1, E_2, E_3,$ and E_4 to the cut after the modifications performed above will not decrease (E_3 will potentially require "flip" operations to ensure its contribution does not go down). This will ensure that $Cut(\mathcal{C}_i) \geq Cut(\mathcal{C}_{i-1})$ (Invariant 2). The four disjoint subsets are defined as follows:

- $E_1 = \{(u, v) | u, v \in \pi_i\}$
- $E_2 = \{(u, v) | w \in \mathcal{L}_i$ and $u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$

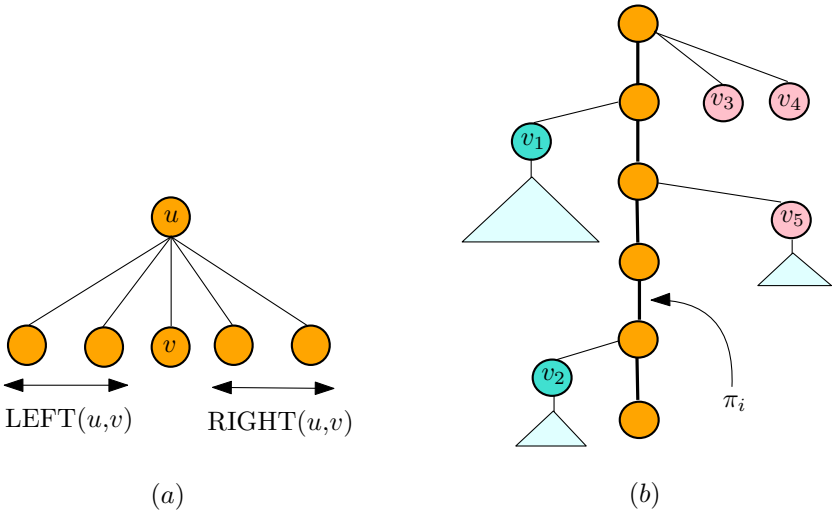

Figure 4 (a) illustrates LEFT(u, v) and RIGHT(u, v), and (b) illustrates $\mathcal{L}_i$ and $\mathcal{R}_i$. In this example, $\mathcal{L}_i = \{v_1, v_2\}$ and $\mathcal{R}_i = \{v_3, v_4, v_5\}$.

- $E_3 = \{(u, v) | w \in \mathcal{R}_i$ and $u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$
- $E_4 = E \setminus E_1 \setminus E_2 \setminus E_3 = \{(u, v) | w \in \mathcal{L}_i \cup \mathcal{R}_i$ and $u, v \in (D(w) \cup \{w\})$ and $v \in A(u)\}$

See Figure 5 for an example. Now we will argue about each subset one after the other.

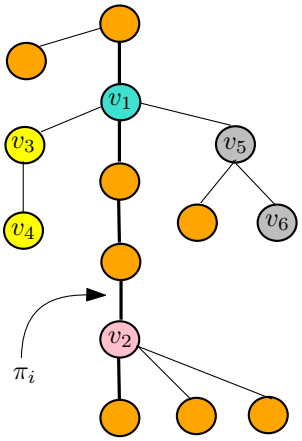

Figure 5 We illustrate via an example some of the pairs in $E_1, E_2, E_3,$ and E_4. For example, $(v_1, v_2) \in E_1, (v_1, v_3) \in E_2, (v_1, v_4) \in E_2, (v_1, v_5) \in E_3, (v_1, v_6) \in E_3, (v_4, v_3) \in E_4, (v_6, v_5) \in E_4$.

▶ **Lemma 4.** *The number of pairs in E_1 which contribute to the cut does not change after the modification.*

Proof. Before and after the modification, the number of pairs of E_1 which contribute to the cut will be equal to $g(\pi_i) \cdot r(\pi_i)$. ◀

▶ **Lemma 5.** *For any $v \in \mathcal{R}_i \cup \mathcal{L}_i$, after performing the modification described above, the number of green ancestors will not decrease.*

Proof. For any $v \in \mathcal{R}_i \cup \mathcal{L}_i$, it is obvious from the above definition that all its ancestors lie on π_i. Let $g(v)$ and $r(v)$ be the number of green and red ancestors of v, respectively, *before* performing the modification. Let $g'(v)$ be the number of green ancestors (excluding v) of v *after* performing the modification.

First, consider the case when $g(\pi_i) \geq r(\pi_i)$. If $r(v) + g(v) \leq g(\pi_i)$, then all the ancestors of v after modification will be colored green and hence, $g'(v) = r(v) + g(v) \geq g(v)$; otherwise, the number of ancestors of v after modification colored green will be equal to $g(\pi_i)$ and hence, $g'(v) = g(\pi_i) \geq g(v)$. Therefore, $g'(v) \geq g(v)$.

Next, consider the case when $r(\pi_i) > g(\pi_i)$. If $r(v) + g(v) \leq r(\pi_i)$, then all the ancestors of v after modification will be colored green and hence, $g'(v) = r(v) + g(v) \geq g(v)$; otherwise, the number of ancestors of v after modification colored green will be equal to $r(\pi_i)$ and hence, $g'(v) = r(\pi_i) > g(\pi_i) \geq g(v)$. Therefore, $g'(v) \geq g(v)$. ◀

▶ **Lemma 6.** *Recall that $E_2 = \{(u, v) | w \in \mathcal{L}_i$ and $u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$. The contribution of E_2 to the cut does not decrease after the modification.*

Proof. Consider any $w \in \mathcal{L}_i$ and let $E_2(w) = \{(u, v) | u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$. First, consider the case where w is colored green before the modification. Then, by invariant 1, since there are no inversions on the path from root to w, all the ancestors of w will be colored green. After the modification, by Lemma 5, it follows that all the ancestors of w will continue to be colored green and hence, the contribution of $E_2(w)$ to the cut does not change.

Next, consider the case where w is colored red before the modification. Then, by invariant 1, all the nodes in $D(w)$ will be colored red. After the modification, by Lemma 5, the number of green ancestors of w will not decrease. Therefore, for any $w \in \mathcal{L}_i$, the contribution of $E_2(w)$ to the cut will not decrease after the modification. ◀

After the modification steps, now we will potentially perform some *flip operations*. The flip operations are needed to handle E_3.

Flip Operations. For any $w \in \mathcal{R}_i$: if $D(w) \cup \{w\}$ has more green colored nodes than red colored nodes, then *flip* all the colors in $D(w) \cup \{w\}$. Formally, each red (resp., green) colored node is now colored green (resp., red).

▶ **Lemma 7.** *Recall that $E_3 = \{(u, v) | w \in \mathcal{R}_i$ and $u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$. The contribution of E_3 to the cut does not decrease after the modification steps and flip operations.*

Proof. Consider any $w \in \mathcal{R}_i$ and let $E_3(w) = \{(u, v) | u \in A(w)$ and $v \in (D(w) \cup \{w\})\}$. Because of flip operations, $D(w) \cup \{w\}$ has either equal or more red colored nodes than green colored nodes. Combining this with Lemma 5 (the number of green ancestors will not come down after modification), we conclude that the contribution of $E_3(w)$ to the cut does not go down after modification and flip operations. ◀

▶ **Lemma 8.** *Recall that $E_4 = E \setminus E_1 \setminus E_2 \setminus E_3 = \{(u, v) | w \in \mathcal{L}_i \cup \mathcal{R}_i$ and $u, v \in (D(w) \cup \{w\})$ and $v \in A(u)\}$. The contribution of E_4 to the cut does not change after the modification steps and the flip operations.*

Proof. Consider any edge $(u, v) \in E_4$. We claim that (u, v) will contribute to the cut after the modification if and only if (u, v) contributed to the cut before the modification: if $w \in \mathcal{R}_i$ and nodes in $D(w) \cup \{w\}$ flipped their colors, then the colors of *both* u and v will change. Otherwise, if $w \in \mathcal{R}_i$ and nodes in $D(w) \cup \{w\}$ did not flip their colors, or if $w \in \mathcal{L}_i$, then the colors of u and v do not change. ◀

▶ **Lemma 9** (Invariant 1). *For all $1 \leq j \leq i$, there are no inversions among nodes on path π_j.*

Proof. It is easy to see that there are no inversions among nodes on path π_i. For any $j < i$, consider the nodes in $\pi_i \cap \pi_j$. Before the modification, there were no inversions on π_j. This implies that either $\pi_i \cap \pi_j$ (a) had all nodes colored green, or (b) had a sequence of green followed by red colored nodes. In case (a), after the modification, all the nodes in $\pi_i \cap \pi_j$ will continue to be colored green. In case (b), after the modification, either all nodes in $\pi_i \cap \pi_j$ will become green, or it will continue to be a sequence of green followed by red colored nodes. In all these cases, there will be no inversions created among nodes on path π_j. ◀

3 A fast implementation

Naively, one can construct the tree representation in $O(n^2)$ time by comparing every pair of objects. In this section, we will construct the tree representation in almost-linear time (in terms of n) for axis-aligned boxes in $3D$ and $2D$. Note that given the tree representation, Algorithm 1 takes only $O(n)$ time: Computing $|D(u)|$, for all $u \in \mathcal{T}$, can be done in $O(n)$ time via bottom-up traversal of $\mathcal{T}$. Analogously, computing $|A(u)|$, for all $u \in \mathcal{T}$, can be done in $O(n)$ time via a top-down traversal of $\mathcal{T}$.

Let $\mathcal{B}$ be a collection of n laminar axis-aligned boxes in $3D$. Now we describe our algorithm to compute the tree representation $\mathcal{T}$ for $\mathcal{B}$. To make the presentation simple, consider a dummy box B_0 which contains all the boxes in $\mathcal{B}$. Add B_0 to $\mathcal{B}$ (this ensures that B_0 is the root of $\mathcal{T}$). Any axis-aligned box in $3D$ has six faces on its boundary. Define the *left face* and the *right face* to be the boundaries parallel to the yz-plane.

We will perform a sweep-plane. Consider a plane ℓ parallel to the yz-plane. Starting from $x = -\infty$ we will move the plane ℓ towards $x = +\infty$. The event points will be the left and the right faces on the boundary of each box in $\mathcal{B}$. Assume that the sweep-plane has reached the x-coordinate x_t, and let ℓ_t be the plane ℓ at x_t. Let $\mathcal{B}_t \subseteq \mathcal{B}$ be the set of boxes intersecting the plane ℓ_t. For every $B \in \mathcal{B}_t$, let $R \leftarrow B \cap \ell_t$, i.e., the projection of B onto the plane ℓ_t which is an axis-aligned rectangle in $2D$.

Along with the sweep-plane, we will maintain a *dynamic vertical ray shooting* data structure. In the *orthogonal* version of dynamic vertical ray shooting problem, the input is a collection of horizontal segments in 2D, and given a query point q' in 2D, the goal is to report the first segment which is hit by the upward ray starting from q'. Updates include insertion and deletion of segments. Let $\mathcal{D}_t$ be the instance of the vertical ray shooting data structure when the sweep-plane is at x_t. For each $B \in \mathcal{B}_t$, the top and the bottom segment of R are maintained in $\mathcal{D}_t$. With the top and the bottom segments of R we will maintain a *label* B and B^{par}, respectively, where B^{par} is the parent of B in $\mathcal{T}$.

▶ **Lemma 10.** *Suppose that the next event point in the sweep-plane is the left face of $B \in \mathcal{B}$. Consider any point $q(q_x, q_y, q_z)$ on the left face of B. Let x_t be the x-coordinate of the sweep-plane just before it reaches the left face of B. If B_q is the label associated with the segment reported by the vertical ray shooting query on $\mathcal{D}_t$ with query point $q'(q_y, q_z)$, then B_q is the parent of B in $\mathcal{T}$.*

Proof. We will consider the following three cases.

- Since B^{par} contains B, it implies that the rectangle $R^{par} \leftarrow B^{par} \cap \ell_t$ will contain q'. Also, all the ancestors of B^{par} in the tree representation will contain q'. However, by the laminar property, notice that a vertical upward ray from q' will hit the top segment of R^{par} before hitting the top segments of its ancestors. See Figure 6(a).

- Consider any box B' which is a child of B^{par} in the tree representation. Since R' does not contain q', then a vertical upward ray from q' will either miss the top and the bottom segments of R', or it will hit the bottom segment of R' before the top segment of R'. See Figure 6(b).
- Consider any box B' which is a descendent of B^{par}, but not a child of B^{par}. Then a vertical upward ray from q' will never hit the bottom or the top segment of B' first. See Figure 6(c).

Based on the above three observations, we conclude that the vertical ray shooting query on $\mathcal{D}_t$ with q' will either report the top segment of R^{par} or a bottom segment corresponding to B^{par}'s children. Note that B^{par} is the label associated with all these segments, and hence, $B_q \leftarrow B^{par}$. ◀

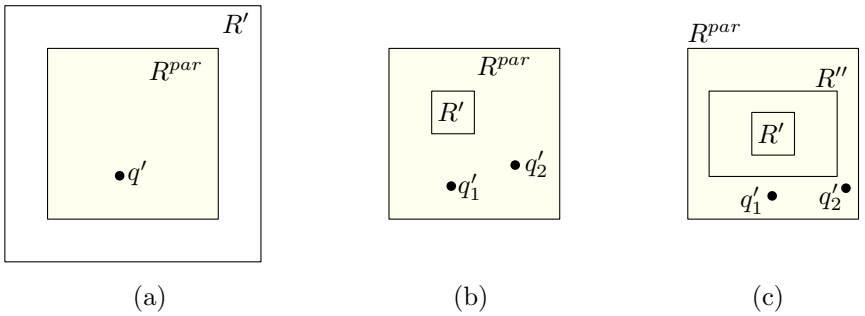

Figure 6 (a) R' is an ancestor of R^{par}, (b) q'_1 hits the bottom segment of R' first and q'_2 hits the top segment of R^{par} first, and (c) Both q'_1 and q'_2 will not hit the bottom segment of R' first. B'' is the ancestor of B' in the tree representation.

Algorithm. Now we are ready to describe the overall algorithm. When the sweep-plane is at x-coordinate $x = x_t$, then the ray shooting data structure $\mathcal{D}_t$ is maintained based on the boxes in $\mathcal{B}_t$. When the sweep-plane's next event point is the left face of a box $B \in \mathcal{B}_t$, then perform the following steps: Using Lemma 10, query $\mathcal{D}_t$ with q' to find the parent B^{par} of B in the tree representation. Update $\mathcal{D}_t$ by inserting the top and the bottom segments of $R \leftarrow B \cap \ell_t$. When the sweep-plane's next event point is the right face of a box $B \in \mathcal{B}_t$, then delete the top and the bottom segments of $R \leftarrow B \cap \ell_t$ from $\mathcal{D}_t$.

Analysis. Sorting the left and the right faces of the boxes in $\mathcal{B}$ can be done in $O(n \log\log n)$ time [20]. At each event point, we will perform at most one vertical ray shooting query. Chan and Tsakalidis [12] present a dynamic data structure for vertical ray shooting with an expected query time of $O(\log n/\log\log n)$ and an amortized update time of $O(\log^{1/2+\varepsilon} n)$. Our algorithm performs $\Theta(n)$ queries and $\Theta(n)$ updates to the ray shooting structure. Therefore, the overall expected time taken is $O(n \log n/\log\log n)$.

▶ **Theorem 11.** *The tree representation for n laminar axis-aligned boxes in 3D can be computed in $O(n \log n/\log\log n)$ expected time.*

▶ **Theorem 12.** *The tree representation for n laminar axis-aligned boxes in 2D can be computed in $O(n \log\log U)$ expected time, where the endpoints of the boxes lie on the integer grid $[U]^2$.*

Proof. We will adapt the algorithm for $3D$ to the $2D$ case. In that case, we will need a dynamic $1D$ point location data structure. By using the data structure of Chan [11], our algorithm can be implemented in $O(n \log \log U)$ expected time. ◀

4 Future work

As discussed in the Introduction, recently max cut on interval graphs has been shown to be NP-hard [1]. For the special case of laminar interval graphs, our works shows that max cut can be computed exactly in almost-linear time (in terms of n). However, for the other special case of max cut on unit interval graphs, the complexity status is still an open problem. Another interesting line of work is to design a polynomial time algorithm for max cut on interval graphs (and other geometric intersection graphs) with an approximation factor better than 0.878.

References

1 Ranendu Adhikary, Kaustav Bose, Satwik Mukherjee, and Bodhayan Roy. Complexity of maximum cut on interval graphs. In *37th International Symposium on Computational Geometry (SoCG)*, volume 189, pages 7:1–7:11, 2021.

2 Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry*, pages 1–56, 1998.

3 Francisco Barahona. The max-cut problem on graphs not contractible to k5. *Operations Research Letters*, 2(3):107–111, 1983.

4 Alexey Barsukov, Kaustav Bose, and Bodhayan Roy. Maximum cut on interval graphs of interval count two is np-complete. *Computing Research Repository (CoRR)*, abs/2203.06630, 2022.

5 Piotr Berman and Marek Karpinski. On some tighter inapproximability results (extended abstract). In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 200–209, 1999.

6 Hans L. Bodlaender, Celina M. H. de Figueiredo, Marisa Gutierrez, Ton Kloks, and Rolf Niedermeier. Simple max-cut for split-indifference graphs and graphs with few p4's. In *Experimental and Efficient Algorithms*, pages 87–99, 2004.

7 Hans L. Bodlaender and Klaus Jansen. On the complexity of the maximum cut problem. *Nordic Journal of Computing*, 7(1):14–31, 2000.

8 Hans L. Bodlaender, Ton Kloks, and Rolf Niedermeier. Simple max-cut for unit interval graphs and graphs with few p4s. *Electronic Notes in Discrete Mathematics*, 3:19–26, 1999.

9 Arman Boyacı, Tinaz Ekim, and Mordechai Shalom. A polynomial-time algorithm for the maximum cardinality cut problem in proper interval graphs. *Information Processing Letters*, 121:29–33, 2017.

10 Arman Boyacı, Tınaz Ekim, and Mordechai Shalom. The maximum cardinality cut problem in co-bipartite chain graphs. *Journal of Combinatorial Optimization*, 35(1):250–265, 2018.

11 Timothy M. Chan. Persistent Predecessor Search and Orthogonal Point Location on the Word RAM. *ACM Trans. Algorithms*, 9(3), 2013.

12 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic Planar Orthogonal Point Location in Sublogarithmic Time. In *34th International Symposium on Computational Geometry (SoCG 2018)*, volume 99, pages 25:1–25:15, 2018.

13 Moses Charikar and Anthony Wirth. Maximizing quadratic programs: Extending grothendieck's inequality. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 54–60, 2004.

14 Celina M. H. de Figueiredo, Alexsander Andrade de Melo, Fabiano de S. Oliveira, and Ana Silva. Maxcut on permutation graphs is np-complete. *Computing Research Repository (CoRR)*, abs/2202.13955, 2022.

15 Celina MH de Figueiredo, Alexsander A de Melo, Fabiano S Oliveira, and Ana Silva. Maximum cut on interval graphs of interval count four is np-complete. *arXiv preprint*, 2020. `arXiv:2012.09804`.

16 Josep Díaz and Marcin Kamiński. Max-cut and max-bisection are NP-hard on unit disk graphs. *Theoretical Computer Science*, 377(1):271–276, 2007.

17 M. X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.

18 Venkatesan Guruswami. Maximum cut on line and total graphs. *Discrete Applied Mathematics*, 92(2):217–221, 1999.

19 F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal of Computing*, 4(3):221–225, 1975.

20 Yijie Han. Deterministic sorting in o($n$loglog$n$) time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

21 Klaus Jansen, Marek Karpinski, Andrzej Lingas, and Eike Seidel. Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs. *SIAM Journal of Computing*, 35(1):110–119, 2005.

22 David S Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 6(3):434–451, 1985.

23 Satyen Kale and C. Seshadhri. Combinatorial approximation algorithms for maxcut using random walks. In *Proceedings of 2nd Symposium on Innovations in Computer Science (ICS)*, pages 367–388, 2011.

24 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum Press, New York, 1972.

25 Jan Kratochvíl, Tomáš Masařík, and Jana Novotná. U-Bubble Model for Mixed Unit Interval Graphs and Its Applications: The MaxCut Problem Revisited. In *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, volume 170, pages 57:1–57:14, 2020.

26 Luca Trevisan. Max cut and the smallest eigenvalue. *SIAM Journal of Computing*, 41(6):1769–1786, 2012.