

Solving Problems in RDV Graphs Faster and Easier

Therese Biedl  

David R. Cheriton School of Computer Science, University of Waterloo, Canada

Günter Rote  

Institut für Informatik, Freie Universität Berlin

Abstract

In interval graphs and many generalizations thereof, problems such as `MATCHING`, `INDEPENDENTSET` and `DOMINATINGSET` can be solved with simple greedy algorithms in linear (i.e., $O(m + n)$) time. However, it is sometimes possible to improve on this runtime if a suitable representation as intersection graph is given; for example it is possible to find a minimum dominating set in an interval graph in $O(n)$ time. We study RDV graphs, a generalization of interval graphs, namely, the intersection graphs of directed paths in a rooted tree. It was known that the above three problems can be solved in $O(n \log n)$ time if a suitably small representation is given. In this paper, we reduce the runtime to $O(n)$. Furthermore, while the previous results used non-trivial operations such as range-minimum queries and ray-shooting queries, we need nothing more complicated than lists and (for `MATCHING`) a `UNION-FIND` data structure.

2012 ACM Subject Classification Mathematics of computing $\rightarrow$ Graph algorithms

Keywords and phrases RDV graph, Dominating Set, Independent Set, Matching, Linear Time

Funding *Therese Biedl*: Supported by NSERC, FRN RGPIN-2020-03958. Research done while the first author visited Freie Universität Berlin during her sabbatical in the winter term 2025/26.

1 Introduction

We study efficient algorithms for graphs that are *RDV graphs*, i.e., vertex-intersection graphs of directed paths in a rooted tree. (Formal definitions are given in Section 2.) This graph class contains interval graphs and is a subclass of the chordal graphs; so any problem that can be solved efficiently in chordal graphs (such as `INDEPENDENTSET`, which is NP-hard in general but can be solved with a linear-time greedy algorithm in chordal graphs [10]) can also be solved efficiently in RDV graphs. In fact, RDV graphs are a subclass of so-called strongly chordal graphs. In strongly chordal graphs, problems such as `DOMINATINGSET` and `MATCHING` can be solved in linear time with a greedy algorithm [6, 8], while no linear-time algorithms are known for chordal graphs (and in fact `DOMINATINGSET` is NP-hard in chordal graphs [4]).

In the above algorithms, “linear” refers to the input size, i.e., the runtime is $O(m + n)$ for a graph with n vertices and m edges. In contrast to this, later papers considered algorithms whose runtime depends only on n , assuming that a suitably small representation of the graph as intersection graph of objects is given. Of particular interest to the current work is a recent string of papers where the first author and Gokhale argued that the runtime of the above greedy algorithms can be reduced to $O(n \log n)$ in RDV graphs [1, 3, 2, 13].¹

Specifically, the runtime of these algorithms is $O(|\mathcal{T}| + n \log n)$, where $\mathcal{T}$ is the tree used for the representation of G . There always exists a representation with $|\mathcal{T}| \in O(n)$, and so the runtime is $O(n \log n)$ if the given representation is small enough (an assumption that we will make throughout).

¹ `MATCHING` is addressed in the conference paper [1], and in its journal version [3] with a simplified approach. `DOMINATINGSET` is addressed in [2], while `INDEPENDENTSET` is addressed only in Gokhale’s Master’s thesis [13], which summarizes all these results.

The crucial idea for these algorithms with runtime depending only on n and not on m is to reframe a given tree representation in geometric terms, namely as an intersection graph of T-shapes. Biedl and Gokhale define for every vertex of G a horizontal and a vertical segment whose union forms a T such that two vertices are adjacent if and only if their Ts intersect. One can then express the operations needed by the greedy algorithm as queries among these segments, and the authors showed that it suffices to use range-maximum queries and orthogonal ray-shooting queries in a (dynamically changing) subset of the segments. Prior results allow these queries to be implemented with runtime $O(\log n)$: McCreight’s priority search trees support range-maximum queries [17], and Giyora and Kaplan gave data structures for orthogonal ray-shooting queries [12]. However, neither of these data structures is straightforward to implement, especially in a dynamic setting.

In this paper, we improve on these algorithms for RDV graphs in three ways. First, we show that the detour into the T-intersection representation is not necessary: everything can be done directly with a bottom-up traversal of the tree $\mathcal{T}$ of the tree representation.

Second, the runtimes are better: we achieve runtime $O(n)$ for all three problems, compared to the previous bound of $O(n \log n)$ [2, 3, 13]. For MATCHING, our runtime even beats the best previously known result for *interval graphs*, which was $O(n \log \log n)$ (Moitra and Johnson [18] with an improvement pointed out by Rhee and Liang [20, Section 5]).

Finally, and this is what we view as our most important contribution, our algorithms are very simple. There is no need for complicated data structures; all the required information can be obtained by storing at each node of $\mathcal{T}$ a few integers or lists or vertex references. (For MATCHING we also use a UNION-FIND data structure, but this is easy to implement if we relax the runtime bound from $O(n)$ to $O(n\alpha(n))$).

Organization of the paper. Section 2 introduces RDV representations and our notation, and Section 3 describes the bottom-up traversal of $\mathcal{T}$ on which our algorithms are based. Sections 4, 5, and 6 give our algorithms for INDEPENDENTSET, DOMINATINGSET, and MATCHING, respectively, each with a small example, and Section 7 mentions open problems. Appendix A contains streamlined pseudocode for all three algorithms, and Appendix B proves that the two greedy algorithms for MATCHING that we build on produce the same matching, and that this matching is maximum.

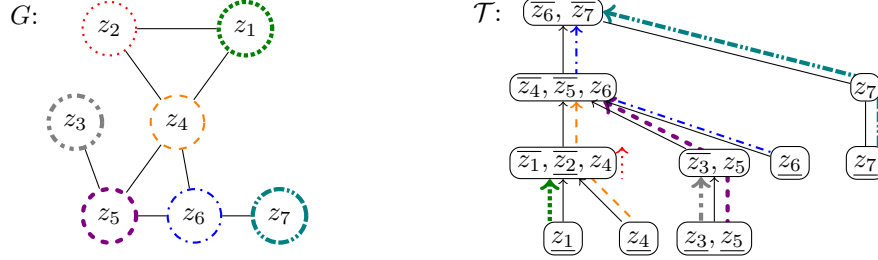
2 Background

We assume familiarity with graphs, and refer e.g. to Diestel’s and Golumbic’s books for basic terminology [7, 14]. In this paper we study graphs G that have a *representation as a vertex-intersection graph of paths in a tree*. This consists of a *host tree* $\mathcal{T}$ and, for each vertex z of G , a path $P(z)$ in $\mathcal{T}$ such that zy is an edge of G if and only if $P(z)$ and $P(y)$ share at least one node of $\mathcal{T}$. By convention, we use the term ‘node’ and early letters $a, b, c, \dots$ of the alphabet for elements of $V(\mathcal{T})$, to distinguish them from the vertices of G for which we use $z, y, x, \dots$.

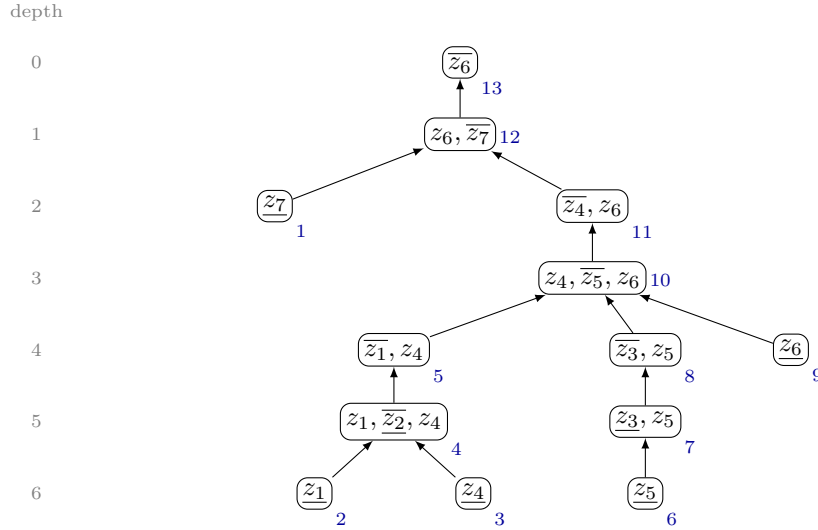
The class of graphs with such a representation is known as the *path graphs*. We study here a graph class that is a proper subclass of the path graphs; it was named by Monma and Wei [19], though studied even earlier e.g. by Gavril [11] and Booth and Johnson [4]:²

² They called RDV graphs *directed path graphs*, but this later came to mean a different graph class, where the edges of $\mathcal{T}$ can be oriented independently without regard to a common root.

► **Definition 1.** A rooted directed path graph (or RDV graph [19]) is a graph that has an RDV representation, i.e., a representation as vertex-intersection graph of directed paths in a rooted tree $\mathcal{T}$ in which all edges are directed towards the root.



■ **Figure 1** An RDV graph G with an RDV representation $\mathcal{T}$. The bags are written in the nodes of $\mathcal{T}$. (Picture taken from [2].)



■ **Figure 2** The running example: an RDV representation of the graph G of Figure 1 that satisfies our assumptions. Each node shows its bag; top nodes are marked by overlining and bottom nodes by underlining the corresponding vertex. The top nodes are distinct, the bottom nodes are distinct, and the root is a top node. The depths of the nodes are shown on the left. Since every node is a top node or a bottom node, we refer to the nodes as $t(z_i)$ and $b(z_i)$. The small numbers give the order in which the nodes are processed in the examples.

Figure 1 illustrates an RDV graph and its RDV representation. RDV graphs can be recognized in polynomial time [11], and along the way, an RDV representation of size $O(n)$ is produced. We use $t(z)$ and $b(z)$ for the *top node* and *bottom node* of the path of a vertex z , respectively. We may without loss of generality assume that $t(z) \neq t(y)$ for any two vertices z, y ; if this is violated (say node a is both $t(z)$ and $t(y)$) then we can subdivide the edge from a to its parent (or add a parent if a is the root) and extend the path of z upward into the new node without affecting the represented graph. Similarly we can require that $b(z) \neq b(y)$ for any two vertices z, y , by adding a new child to $b(z)$ if required. Finally we may assume that the root is a top node, for if it is not then no path uses it, and so we can simply contract one of its children with the root without affecting the represented graph.

These assumptions make the statements of the algorithms more convenient. Figure 2 shows an RDV representation of the graph of Figure 1 that satisfies all of them; we will use it as a running example.³

We assume that every vertex z can look up nodes $t(z)$ and $b(z)$ in constant time. We also prepare $\mathcal{T}$ such that each node a can do the following in constant time:

1. Look up $\text{depth}(a)$, which is the distance of a from the root of $\mathcal{T}$.
2. Test whether a is the top node of some $z \in V(G)$, and if so, look up this (unique) vertex z .
3. Test whether a is the bottom node of some $z \in V(G)$, and if so, look up this (unique) vertex z .

For any $a \in V(\mathcal{T})$, we write $\beta(a)$ for the set of vertices whose paths contain a , i.e., $\beta(a) = \{z \in V(G) : a \in P(z)\}$. We call $\beta(a)$ the *bag* of a , a name inspired by the fact that $\mathcal{T}$ and the assignment of paths is a special case of a so-called tree decomposition (see e.g. [7]) where the corresponding concept is called a bag. We will need a simple observation.

► **Observation 2.** *Consider a vertex z and its top node $a = t(z)$. Every neighbor w of z either is in $\beta(a)$, or $t(w)$ is a strict descendant of a .*

Proof. Assume that $w \notin \beta(a)$, so $P(w)$ does not contain a . But $P(w)$ must share a node with the path $P(z)$, say node $b \neq a$. This must be a descendant of a since a is the top node of $P(z)$, and it is a *strict* descendant since $b \neq a$. So $P(w)$ contains strict descendants of a , but does not contain a ; thus $t(w)$ must be a strict descendant of a . ◀

We will use the common notation $N[z]$ for the closed neighborhood of a vertex z , consisting of z together with its adjacent vertices. For a set of vertices W , $N[W]$ denotes the union of $N[z]$ over all $z \in W$. We study the following three problems:

- **INDEPENDENTSET:** Given a graph G , find a maximum vertex set I that is *independent*, i.e., no edge connects two vertices in I .
- **DOMINATINGSET:** Given a graph G , find a minimum vertex set D that is *dominating*, i.e., each vertex of G that is not in D has a neighbor in D , or more formally, $N[D] = V(G)$.
- **MATCHING:** Given a graph G , find a maximum edge set M such that no two edges of M have a common endpoint.

We will not review the extensive history of these three problems, and what is known in terms of algorithms and/or NP-hardness, except for graph classes that are closely related to RDV graphs, where such a review will be given in the relevant section below.

3 The Common Structure of Our Algorithms

Our algorithms for **INDEPENDENTSET** and **DOMINATINGSET** have a common basic structure: They perform a bottom-up traversal of the tree $\mathcal{T}$. At each node, the same 4-step procedure is used. Our algorithm for **MATCHING** also processes the nodes of $\mathcal{T}$ from the leaves to the root, but it deviates from this scheme: it does not work with the bags, but with a **UNION-FIND** data structure on the nodes of $\mathcal{T}$ (Section 6). We illustrate this paradigm with a skeleton that makes the bags $\beta(a)$ available when needed.

³ We could also stipulate that no node is both a top node and a bottom node, but we do not need this. A degenerate path consisting of a single node is also permitted.

Processing node a . We assume that all strict descendants of a have already been processed.

1. (Accumulate the data from the children.)
 - If a is a leaf, initialize an empty list $B(a)$.
 - Otherwise, let $B(a)$ be the concatenation of lists $B(c)$ over all children c of a .
2. (Update if a path starts.) If a is a bottom node, say $a = b(z)$:
 - Add z to $B(a)$.
3. (Make decisions.)
 - ... [At this point, $B(a) = \beta(a)$.]
4. (Update if a path ends.) If a is a top node, say $a = t(z)$:
 - Remove z from $B(a)$.

► **Lemma 3.** *For any node a , the set $B(a)$ is always a subset of $\beta(a)$, and at the beginning of Step 3 when a is processed, $B(a) = \beta(a)$.*

Proof. In fact, the manipulations of $B(a)$ in Steps 1, 2, and 4 play together to ensure that the correct set $B(a)$ is built inductively. In Step 4, a vertex z with $t(z) = a$ is removed from $B(a)$ in preparation for building the set B for the parent. Thus in Step 1 the set at each child contains all those vertices whose paths contain the child and continue upward. So all these paths continue into a , hence the vertices are in $\beta(a)$. Clearly the vertex added in Step 2 (if any) is also in $\beta(a)$. Vice versa, any vertex in $\beta(a)$ corresponds to a path that contains a . This path either had its bottom node at a (such vertices are added in Step 2), or it contains one of the children (and hence the vertex is added in Step 1). ◀

As it happens, the later algorithms will not actually need the sets $B(a)$, but they will compute subsets of $B(a)$, and we shall use $B(a)$ for their correctness arguments.

Leaf-to-root order. The only required condition on the order of processing is that a child comes before its parent. Any order that is consistent with this can be used in our algorithms, for example postorder or reverse level-order.

Optimality. Our three algorithms establish optimality in different ways. For INDEPENDENTSET, the algorithm computes, along with the independent set, a clique cover of the same size, which certifies optimality. For DOMINATINGSET, we show that our algorithm computes the same set as the greedy algorithm of Booth and Johnson [4], and rely on their optimality proof. For MATCHING, we show that our algorithm computes the same matching as a known greedy algorithm, whose optimality follows from the general results of Dahlhaus and Karpin-ski [6] about strongly chordal graphs; Appendix B.1 gives an independent self-contained proof for our setting.

4 Maximum Independent Set I

It is well known that in a chordal graph, a maximum independent set can be found with a simple linear-time greedy algorithm executed on a perfect elimination order (see Gavril’s original paper [10] or Golumbic’s monograph [14] for details). In an RDV graph, the runtime can be reduced to $O(n \log n)$ by employing suitable data structures [13, Thm. 5.2.2]. For interval graphs where the interval endpoints are given in sorted order (this corresponds to our assumption that the tree representation $\mathcal{T}$ is given), a maximum independent set I can be found by a simple greedy algorithm in $O(n)$ time [15].

In this section we give an algorithm that solves INDEPENDENTSET in an RDV graph in $O(n)$ time. Our algorithm will perform the same steps as the greedy algorithm, but rather

than showing this we will argue directly that it computes a maximum independent set. In accordance with the paradigm in Section 3, the algorithm performs a bottom-up traversal of $\mathcal{T}$ and incrementally builds an independent set I .

We maintain a label $state(z)$ for every vertex z with the following meaning:

(I1) Every vertex z in $N[I]$ has $state(z) = \mathbf{blocked}$; all other vertices have $state(z) = \mathbf{free}$. We also maintain a list $L(a)$ for every node a , with the following invariant:

(I2) $L(a) = \{z \in B(a) : state(z) = \mathbf{free}\}$,

where $B(a)$ is the set introduced in Section 3; recall in particular that $B(a) = \beta(a)$ during Step 3 when a is processed. For reference, we include the manipulations of the sets $B(a)$ from the algorithm of Section 3 as comments.

As an optimality certificate, we store for each $z \in I$ a vertex set $clique(z)$ with the following property (as the name suggests):

(I3) $clique(z)$ forms a clique.

Initialization. Initialize an empty set I of vertices. For every vertex $z \in V(G)$, initialize $state(z) := \mathbf{free}$.

Processing node a . We assume that all strict descendants of a have already been processed.

1. (Accumulate the data from the children.)
 - // Form $B(a)$ by concatenating the lists $B(c)$ of the children c of a .
 - Form $L(a)$ by concatenating the lists $L(c)$ of the children c of a .
2. (Update if a path starts.) If a is a bottom node, say $a = b(z)$:
 - // Add z to $B(a)$
 - Add z to $L(a)$ // $state(z) = \mathbf{free}$ at this point.
3. (Add to solution if possible.) If a is a top node, say $a = t(z)$, and $state(z) = \mathbf{free}$:
 - Add z to I .
 - // At this point, $L(a) = \{z \in \beta(a) : state(z) = \mathbf{free}\}$.
 - For every vertex w in $L(a)$, set $state(w) := \mathbf{blocked}$. // This includes the vertex z .
 - Set $clique(z) := L(a)$, and clear $L(a)$ to be empty.
4. (Update if a path ends.) If a is a top node, say $a = t(z)$:
 - // Remove z from $B(a)$. // No update is needed for $L(a)$.

After processing the root node, we return I as the maximum independent set.

Correctness. Invariant (I2). In Steps 1 and 2, the set $L(a)$ is constructed and updated in accordance with the defining condition of $L(a)$ that the state should be **free**. In fact, in Step 2, $state(z)$ must be **free**, because the state has not changed since initialization. Hence the addition of z to $L(a)$ is correct. In Step 4, it would be necessary to remove z from $L(a)$ in case $state(z) = \mathbf{free}$. However, in this case, $L(a)$ has already been emptied in Step 3. Consequently, Step 4 is not necessary.

We make an easy but important observation.

(I4) All vertices z for which $t(z)$ has been processed completely have $state(z) = \mathbf{blocked}$.

Proof: When processing $t(z)$, the state of z is either **free** or **blocked**. If z is **free**, Step 3 adds it to I , which makes it **blocked**; so either way $state(z) = \mathbf{blocked}$ afterwards.

We now prove the crucial property that the $state$ labels are maintained correctly (Invariant (I1)): We mark vertices as **blocked** if and only if they were in $L(a)$ during Step 3 and we added the vertex $z \in L(a)$ to I . Since $L(a)$ forms a clique, all these vertices are now in $N[z] \subseteq N[I]$, and therefore their classification as **blocked** is correct.

We still have to show that every vertex $w \in N[z]$ is **blocked** after Step 3 adds z to I :

- If $w \notin \beta(a)$, then by Observation 2 its top node $t(w)$ is a strict descendant of a and has been processed before; by (I4) this means that w is already **blocked**.
- If $w \in \beta(a)$ and is still **free** when processing $a = t(z)$, then by (I2) w is in $L(a)$, and we mark it as **blocked**.

We now turn to the sets $\text{clique}(z)$ (Invariant (I3)): We set $\text{clique}(z) = L(t(z))$ when processing $a = t(z)$. This is a clique since $L(a) \subseteq B(a) \subseteq \beta(a)$, and $\beta(a)$ forms a clique.

We make an additional observation, which will be important for the optimality proof:

(I5) Every vertex w with $\text{state}(w) = \text{blocked}$ is in $\text{clique}(z)$ for a unique $z \in I$.

Proof: Inspection of Step 3 shows that a vertex y is added to some $\text{clique}(z)$ exactly when its state changes from **free** to **blocked**. Since states never change back to **free**, y will never be added to another set $\text{clique}(z')$.

Runtime. We represent the lists $L(a)$ by linked lists, and form list $L(a)$ in Step 1 by concatenation in constant time per child. The lists $L(c)$ of the children are no longer available afterwards, but are also no longer needed. The transfer of the list $L(a)$ to $\text{clique}(z)$ in Step 3 can also be carried out in constant time since $L(a)$ is simultaneously emptied.

► **Theorem 4.** *Let G be an n -vertex RDV graph with a given RDV representation of size $O(n)$. Then the algorithm computes a maximum independent set I of G together with a clique cover of size $|I|$ in $O(n)$ time.*

Proof. We only add a vertex z to I if $\text{state}(z) = \text{free}$, and by Invariant (I1) this means that z is not adjacent to any vertex in I . Hence I is an independent set.

By property (I4), all vertices are **blocked** at the end of the algorithm, and Invariant (I5) in connection with Invariant (I3) implies that the family $\{\text{clique}(z) : z \in I\}$ forms a *clique cover*, i.e., it consists of vertex-disjoint cliques that together cover all vertices of G . By construction, the clique cover has size $|I|$.

A clique cover is a certificate of optimality for I : No independent set can contain more than one vertex in a clique, so no independent set can be larger than any clique cover. So I is a maximum independent set. The existence of a clique cover of size $|I|$ is in line with the fact that RDV graphs are perfect graphs.

The runtime is $O(n)$ since we spend constant time per node and per child and $|\mathcal{T}| \in O(n)$. ◀

A streamlined version of the algorithm (without the clique cover) is shown in Algorithm 5 in Appendix A.1.

Example. Consider the running example of Figure 2, with the nodes processed in the order $b(z_7), b(z_1), b(z_4), t(z_2), t(z_1), b(z_5), b(z_3), t(z_3), b(z_6), t(z_5), t(z_4), t(z_7), t(z_6)$. The first top node to be processed is $t(z_2)$, where $L(t(z_2)) = \{z_1, z_2, z_4\}$; so z_2 is added to I , all three vertices become **blocked**, and $\text{clique}(z_2) = \{z_1, z_2, z_4\}$. At $t(z_1)$, vertex z_1 is **blocked**, and nothing happens. At $t(z_3)$, we have $L(t(z_3)) = \{z_3, z_5\}$, so z_3 is added to I with $\text{clique}(z_3) = \{z_3, z_5\}$. At $t(z_5)$ and $t(z_4)$, the vertices z_5 and z_4 are **blocked**. At $t(z_7)$, we have $L(t(z_7)) = \{z_6, z_7\}$, so z_7 is added to I with $\text{clique}(z_7) = \{z_6, z_7\}$; at the root $t(z_6)$, vertex z_6 is already **blocked**. The result is $I = \{z_2, z_3, z_7\}$, together with the clique cover $\{z_1, z_2, z_4\}, \{z_3, z_5\}, \{z_6, z_7\}$ of the same size.

5 Minimum Dominating Set D

Booth and Johnson [4] appear to have been the first to study DOMINATINGSET in RDV graphs. They gave an algorithm (repeated in Algorithm 1) that is a bottom-up traversal of the tree $\mathcal{T}$ of the representation. They argue that it finds a minimum dominating set, and that it can be implemented with linear (meaning $O(m + n)$) runtime. Gokhale restated this algorithm as a greedy algorithm based on a vertex ordering, which then works whenever this ordering is a so-called strong perfect elimination order [13]. With this reframing, the algorithm can then be implemented with $O(n \log n)$ runtime in RDV graphs [2]. Cheng, Kaminski and Zaks [5] claimed an $O(n)$ time algorithm for DOMINATINGSET in a subclass of path graphs unrelated to RDV graphs. For this, they used as a subroutine an $O(n)$ time algorithm for DOMINATINGSET in interval graphs. Unfortunately, none of the references that they give seems to contain such an algorithm; but $O(n)$ runtime for interval graphs can also be achieved by modifying the techniques for RDV graphs slightly [2].

■ **Algorithm 1** Greedy algorithm for DOMINATINGSET from [4]

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

- 1 initialize $D := \emptyset$, mark all vertices as **undominated**
- 2 **for** each node a of $\mathcal{T}$, in leaf-to-root order **do**
- 3 **if** $t(z) = a$ for some vertex z and z is **undominated** **then**
- 4 let $y \in \beta(a)$ be the vertex with highest top node, i.e., minimizing $\text{depth}(t(y))$
- 5 add y to D and mark all its neighbors as **dominated**

Result: D

The algorithm that we give below for DOMINATINGSET in RDV graphs is the same in spirit as Algorithm 1, but we pre-compute some of the information so that the runtime is reduced from the (obvious) $O(m + n)$ to $O(n)$. Like Algorithm 1, the algorithm performs a bottom-up traversal of $\mathcal{T}$ and incrementally builds a dominating set. To facilitate the correctness argument, we distribute this dominating set among the nodes, and associate with every node a “local” dominating set $D(a)$; eventually these will all get merged into one set that we return. Additionally the algorithm keeps track of three vertices $z_{\text{high}}, z_{\text{dom}}, z_{\text{undom}}$, and three integers h, d, u at each node a of $\mathcal{T}$. The intended meaning is that the following hold at the entry to Step 3 of processing a :

- (D1) $z_{\text{high}}(a)$ is the vertex in bag $\beta(a)$ whose top node is highest, i.e., minimizes the depth of the top node.
- (D2) $z_{\text{dom}}(a)$ is a dominating vertex in the bag (i.e., $z_{\text{dom}}(a) \in \beta(a) \cap D(a)$), and among those, the one whose top node is highest.
- (D3) $z_{\text{undom}}(a)$ is an undominated vertex in the bag (i.e., $z_{\text{undom}}(a) \in \beta(a) \setminus N[D(a)]$), and among those, the one whose top node is lowest.
- (D4) $h(a), d(a)$, and $u(a)$ are the depths of the top nodes of $z_{\text{high}}(a), z_{\text{dom}}(a)$ and $z_{\text{undom}}(a)$, respectively.

If any of these vertices does not exist, we set $z_{\text{high}}(a), z_{\text{dom}}(a)$, or $z_{\text{undom}}(a)$ to **nil**, and accordingly set $h(a) = \infty, d(a) = \infty$, or $u(a) = -\infty$.⁴

Processing node a . We assume that all strict descendants of a have already been processed.

⁴ We write $\pm\infty$ for convenience; it suffices to use values outside the range of depths of nodes of $\mathcal{T}$.

1. (Accumulate the data from the children.) If a is a leaf:
 - Initialize $D(a)$ as an empty set.
 - Set $z_{\text{high}}(a) := z_{\text{dom}}(a) := z_{\text{undom}}(a) := \text{nil}$, $h(a) := d(a) := \infty$, and $u(a) := -\infty$.
 Otherwise (a has children):
 - h. Let $c^* := \operatorname{argmin}\{h(c) : c \text{ is a child of } a\}$. Set $z_{\text{high}}(a) := z_{\text{high}}(c^*)$ and $h(a) := h(c^*)$.
 - D. Form $D(a)$ by concatenating the lists $D(c)$ of the children c of a .
 - d. Let $c^* := \operatorname{argmin}\{d(c) : c \text{ is a child of } a\}$. Set $z_{\text{dom}}(a) := z_{\text{dom}}(c^*)$ and $d(a) := d(c^*)$.
 - u. If $d(a) = \infty$,
 - set $c^* := \operatorname{argmax}\{u(c) : c \text{ is a child of } a\}$, $z_{\text{undom}}(a) := z_{\text{undom}}(c^*)$, $u(a) := u(c^*)$.
 - Otherwise set $z_{\text{undom}}(a) := \text{nil}$, $u(a) := -\infty$.
 2. (Update if a path starts.) If a is a bottom node, say $a = b(z)$:
 - h. If $\text{depth}(t(z)) < h(a)$ update $z_{\text{high}}(a) := z$, $h(a) := \text{depth}(t(z))$.
 - D. // No update is necessary for $D(a)$.
 - d. // No update is necessary for $z_{\text{dom}}(a)$ and $d(a)$.
 - u. If $d(a) = \infty$ and $\text{depth}(t(z)) > u(a)$ update $z_{\text{undom}}(a) := z$, $u(a) := \text{depth}(t(z))$.
 3. (Add to output if needed.) If $u(a) = \text{depth}(a)$:

(In this case, $z := z_{\text{undom}}(a)$ has $t(z) = a$ and is not dominated by $D(a)$.)

 - h. // No update is necessary for $z_{\text{high}}(a)$ and $h(a)$.
 - D. Add $z_{\text{high}}(a)$ to $D(a)$.
 - d. Update $z_{\text{dom}}(a) := z_{\text{high}}(a)$, $d(a) := h(a)$.
 - u. Update $z_{\text{undom}}(a) := \text{nil}$, $u(a) := -\infty$.
 4. (Update if a path ends.) If a is a top node:
 - h. If $h(a) = \text{depth}(a)$ then set $z_{\text{high}}(a) := \text{nil}$, $h(a) := \infty$.
 - D. // No update is necessary for $D(a)$.
 - d. If $d(a) = \text{depth}(a)$ then set $z_{\text{dom}}(a) := \text{nil}$, $d(a) := \infty$.
 - u. // No update is necessary for $z_{\text{undom}}(a)$ and $u(a)$.
- After processing the root node a , we return $D(a)$ as the minimum dominating set.

Clearly the time to process a node of $\mathcal{T}$ is proportional to its number of children, so the runtime of this algorithm is $O(|\mathcal{T}| + n) = O(n)$.

Correctness. The proof that the invariants are maintained is straightforward. Directly from the algorithm it is clear that Invariant (D4) holds, i.e., $h(a)$, $d(a)$, and $u(a)$ are the depths of the top nodes of $z_{\text{high}}(a)$, $z_{\text{dom}}(a)$ and $z_{\text{undom}}(a)$, respectively. To prove the other invariants, we use again the set $B(a)$ defined in Section 3, which equals $\beta(a)$ during Step 3 (Lemma 3).

Invariant (D1): It suffices to show that $z_{\text{high}}(a) = \operatorname{argmin}\{\text{depth}(t(z)) : z \in B(a)\}$ at all times. We add to $B(a)$ during Steps 1 and 2, by either combining the sets of the children or adding a vertex whose bottom node is a . During both steps we update $z_{\text{high}}(a)$, if needed, to take into account the vertices that were newly added to $B(a)$. Step 3 does not change the bag. During Step 4, if we had *any* vertex y left in $B(a)$ after removing the one whose path ends, then $t(y)$ would be above a since top nodes are distinct. So $h(a) = \text{depth}(a)$ implies that $B(a)$ is now empty, justifying that we set $z_{\text{high}}(a) = \text{nil}$.⁵

Invariant (D2): It suffices to show that $z_{\text{dom}}(a) = \operatorname{argmin}\{\text{depth}(t(z)) : z \in B(a) \cap D(a)\}$ at all times. We add to $B(a)$ during Steps 1 and 2. During Step 1 we take all added vertices into account. During Step 2, the newly added vertex z is not in $D(a)$, so no update is needed.

⁵ In particular this implies that no path in the subtree rooted at a intersects a path in the rest of $\mathcal{T}$. Thus in a connected graph this update step is used only when we have processed the entire graph, and could be omitted.

During Step 3, since some vertex in $\beta(a)$ was undominated, none of the vertices in $\beta(a)$ were in $D(a)$ previously, so the newly added $z_{\text{high}}(a)$ is the *only* candidate for $z_{\text{dom}}(a)$. During Step 4, as for (D1) one argues that $d(a) = \text{depth}(a)$ implies that $B(a) \cap D(a)$ is now empty, justifying that we set $z_{\text{dom}}(a) = \text{nil}$.

Invariant (D3): During Steps 1 and 2, if $d(a) < \infty$ then we must have at least one dominating vertex in $B(a)$. This dominates *all* vertices in $B(a)$ and is in $D(a)$, so $z_{\text{undom}}(a)$ is undefined. The same argument applies to Step 3, where $d(a) < \infty$ holds always. We claim that during Step 4, no update is needed to $z_{\text{undom}}(a)$. To see this, assume that $a = t(z)$ is a top node and $z = z_{\text{undom}}(a)$, so that the removal of z from $B(a)$ would necessitate finding a new vertex $z_{\text{undom}}(a)$. This means that $u(a) = \text{depth}(t(z)) = \text{depth}(a)$. But then we would have acted in Step 3, ensuring that z becomes dominated and updating $z_{\text{undom}}(a)$ accordingly.

Finally, we prove that our algorithm computes the same dominating set as Algorithm 1. More specifically, we claim that we add a vertex to $D(a)$ in Step 3 if and only if Algorithm 1 adds a vertex to D . (In this case, both algorithms obviously add the same vertex, $z_{\text{high}}(a)$.)

We shall show that the following four statements are equivalent:

1. Our algorithm adds a vertex to $D(a)$ in Step 3.
2. The vertex z with $t(z) = a$ is not dominated by $D(a)$.
3. The vertex z with $t(z) = a$ is not dominated by the global set D .
4. Algorithm 1 adds a vertex to D when processing a .

The equivalence between 3 and 4 follows directly from the description of Algorithm 1. However, the equivalence between 1 and 2 is not immediately apparent from the test in Step 3. It can be seen as follows:

A vertex is added to $D(a)$ when we are processing a and $u(a) = \text{depth}(a)$. Since $u(a) > -\infty$, the vertex $z := z_{\text{undom}}(a) \neq \text{nil}$ exists, and since $u(a) = \text{depth}(a)$ we have $t(z) = a$ by (D4). By (D3) we conclude that z is undominated by $D(a)$. Conversely, if z with $t(z) = a$ is undominated by $D(a)$, it follows immediately from (D3) and (D4) that $z_{\text{undom}}(a) = z$ and $u(a) = \text{depth}(t(z)) = \text{depth}(a)$, and hence the test in Step 3 will be positive.

For this, we need another observation:

(D5) For every vertex $z \in D(a)$, the path $P(z)$ contains nodes in the subtree rooted at a . Inductively, this holds after concatenating $D(a)$ from the sets of the children in Step 1. The only vertex added to $D(a)$ is $z_{\text{high}}(a)$ in Step 3, which is in $\beta(a)$ by (D1). Its path therefore contains a .

To prove the equivalence between 2 and 3, we assume inductively that $\bigcup_c D(c)$ of our algorithm equals D for Algorithm 1. What remains to prove is that if z is undominated by $D(a)$, it is also undominated by the global set D . If z were dominated by, say, $y \in D$, then $y \in D(b)$ for some node b that is neither an ancestor nor a descendant of a . By (D5), $P(y)$ contains nodes in the subtree rooted at b . But no such vertex y can be adjacent to z , where $a = t(z)$, due to Observation 2. This contradiction proves our claim.

So the two algorithms compute the same dominating set, and since Algorithm 1 finds a minimum dominating set [4], so does ours.

► **Theorem 5.** *Let G be an n -vertex RDV graph with a given RDV representation of size $O(n)$. Then we can find a minimum dominating set of G in $O(n)$ time.* ◀

We note that our algorithm can be slightly simplified: the vertex references z_{dom} and z_{undom} are not needed as long as we store the depths of their top nodes, i.e., d and u . See also Appendix A.2.

Example. Table 1 shows the run of the algorithm on the running example of Figure 2, with the nodes processed in the same order as in the example for INDEPENDENTSET. At $t(z_2)$, the bag contains the undominated vertex z_2 itself, so $u(t(z_2)) = 5 = \text{depth}(t(z_2))$, and the vertex $z_{\text{high}} = z_4$, whose top node is highest in the bag, is added to D . Similarly, z_5 is added at $t(z_3)$. At $t(z_4)$, the dominating vertex z_4 leaves the bag, so Step 4 resets z_{dom} to nil and d to ∞ . Hence, at $t(z_7)$, the vertex z_7 is undominated, and z_6 is added. The result is $D = \{z_4, z_5, z_6\}$. It is a minimum dominating set: the closed neighborhoods $N[z_2] = \{z_1, z_2, z_4\}$, $N[z_3] = \{z_3, z_5\}$, and $N[z_7] = \{z_6, z_7\}$ are pairwise disjoint, so every dominating set has at least three vertices.

iteration	node a	$\text{depth}(a)$	bag $\beta(a)$	$h(a)$	$d(a)$	$u(a)$	added to D
1	$b(z_7)$	2	z_7	1 (z_7)	∞	1 (z_7)	
2	$b(z_1)$	6	z_1	4 (z_1)	∞	4 (z_1)	
3	$b(z_4)$	6	z_4	2 (z_4)	∞	2 (z_4)	
4	$t(z_2)$	5	z_1, z_2, z_4	2 (z_4)	∞	5 (z_2)	z_4
5	$t(z_1)$	4	z_1, z_4	2 (z_4)	2 (z_4)	$-\infty$	
6	$b(z_5)$	6	z_5	3 (z_5)	∞	3 (z_5)	
7	$b(z_3)$	5	z_3, z_5	3 (z_5)	∞	4 (z_3)	
8	$t(z_3)$	4	z_3, z_5	3 (z_5)	∞	4 (z_3)	z_5
9	$b(z_6)$	4	z_6	0 (z_6)	∞	0 (z_6)	
10	$t(z_5)$	3	z_4, z_5, z_6	0 (z_6)	2 (z_4)	$-\infty$	
11	$t(z_4)$	2	z_4, z_6	0 (z_6)	2 (z_4)	$-\infty$	
12	$t(z_7)$	1	z_6, z_7	0 (z_6)	∞	1 (z_7)	z_6
13	$t(z_6)$	0	z_6	0 (z_6)	0 (z_6)	$-\infty$	

Table 1 The algorithm for DOMINATINGSET on the running example of Figure 2. The values h , d , and u are those at the beginning of Step 3; the corresponding vertices z_{high} , z_{dom} , and z_{undom} are given in parentheses.

6 Matching

There is a well-known greedy algorithm for MATCHING (repeated below as Algorithm 2). It uses the notion of a *free* vertex, which is any vertex that is not incident to any edge in the (current) matching. In an RDV graph, the greedy algorithm computes a maximum matching. (In fact, it does so in any strongly chordal graph if one uses a strong perfect elimination order as vertex ordering [6].) In an interval graph, the runtime can be improved to $O(n \log n)$ by reformulating the algorithm a bit [18]. By replacing ‘binary search tree’ in this reformulation with ‘van Emde Boas tree’, the runtime can be reduced further to $O(n \log \log n)$ [20]. A runtime of $O(n \log \log n)$ can also be achieved (with a different approach) for circular arc graphs [16] and permutation graphs [20]. For RDV graphs, the greedy algorithm can be implemented with $O(n \log n)$ runtime [3].

In this section, we give a faster algorithm for MATCHING in RDV graphs. At first glance, one might be tempted to do this by “translating” Algorithm 2 into a traversal in $\mathcal{T}$: Traverse the nodes in leaf-to-root order, act on z_i when we reach $t(z_i)$, and search for z_q among the free vertices of $\beta(t(z_i))$. However, searching through this entire set would take too long. Pre-computation of z_q (similar to what we did with z_{undom}) does not work either, because once z_q is matched, we would need to update this reference (for passing to the parent), and

■ **Algorithm 2** Classic greedy algorithm for MATCHING (see e.g. [6])

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

- 1 let $z_1, \dots, z_n$ be any leaf-to-root ordering of $V(G)$
- 2 initialize the matching $M := \emptyset$
- 3 **for** $i = 1, \dots, n$ **do**
- 4 **if** z_i is free and has free neighbors **then**
- 5 among all free neighbors of z_i , let z_q be the one that minimizes q // ($q > i$)
- 6 add $z_i z_q$ to M

Result: M

for this would again have to inspect all free vertices of $\beta(t(z_i))$.⁶ With a priority queue this could be done in $O(\log n)$ time per node (details omitted), but we want to be faster than that.

We therefore use an entirely different approach based on a reformulation of the greedy algorithm that delays adding a matching edge until we have seen the second endpoint of it. See Algorithm 3 for this reformulation, which was given for an interval representation by Liang and Rhee [16] and stated using a vertex ordering by Biedl and Gokhale [1]. Neither reference formally proved that the two algorithms produce the same matching; for completeness we give such a proof in Appendix B, together with a self-contained optimality proof.

■ **Algorithm 3** The delayed greedy algorithm for MATCHING [16]

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

- 1 let $z_1, \dots, z_n$ be any leaf-to-root ordering of $V(G)$
- 2 initialize the matching $M := \emptyset$
- 3 **for** $q = 1, \dots, n$ **do**
- 4 **if** z_q has free neighbors z_i with $i < q$ **then**
- 5 among such neighbors of z_q , let z_i be the one that minimizes i // ($i < q$)
- 6 add $z_i z_q$ to the matching M

Result: M

The crucial operation for carrying out Algorithm 3 is to find, among the free neighbors of z_q , the neighbor z_i with the smallest i , i.e., with the top node $t(z_i)$ as low as possible, or to detect that there are no such neighbors with $i < q$. In terms of the tree $\mathcal{T}$, this operation can be interpreted as follows: We start from $b(z_q)$ and walk upwards until we hit the first node that is a top node of a free vertex. We denote this operation of “searching for the first top node of a free vertex, starting from a node c and going upward” by $search(c)$.

If this walk reaches $t(z_q)$ without finding such a node, then z_q has no free neighbor z_i with $i < q$ since the top node of such a z_i would have to be a strict descendant of $t(z_q)$ by Observation 2, but an ancestor of $b(z_q)$ since $z_i z_q$ is an edge. Vice versa, if z_q has no free neighbor z_i with $i < q$, then $search(b(z_q))$ will stop at $t(z_q)$ itself, because z_q is free and thus $t(z_q)$ is a candidate node for stopping. On the other hand, if the walk stops at a top node $t(z_i)$ of some free vertex z_i with $i < q$, then this must be the desired matching partner according to Algorithm 3, since the top nodes $t(z_i)$ along the path from $b(z_q)$ to $t(z_q)$ appear

⁶ This was not a problem in our algorithm for DOMINATINGSET, because the vertex added to D dominates every vertex in the bag and hence the updated z_{undom} is simply **nil**.

in increasing order of i . In summary, Algorithm 3 is equivalent to Algorithm 4 given below.

■ **Algorithm 4** The delayed greedy algorithm for MATCHING in terms of the *search*-operation

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

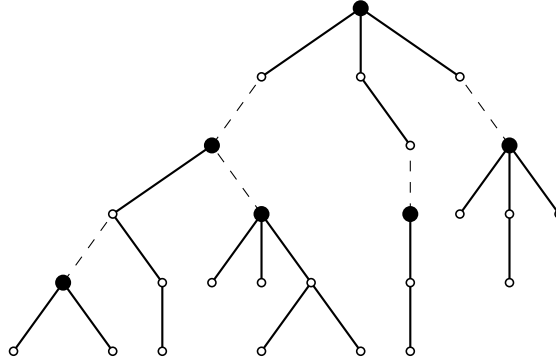
```

1 initialize the matching  $M := \emptyset$ 
2 for each node  $a \in V(\mathcal{T})$ , in leaf-to-root order do
3   if  $a = t(z)$  for some vertex  $z$  then
4     compute  $c := \text{search}(b(z))$ 
5     if  $c \neq a$  then
6       let  $y$  be the vertex with  $t(y) = c$ 
7       add edge  $yz$  to  $M$ 

```

Result: M

So we now need a data structure that facilitates $\text{search}(a)$. An idea for this is illustrated in Figure 3. Here the top nodes of free vertices are marked as thick bullets. Severing the edges between these nodes and their parent nodes (dashed edges) splits the tree into connected components. For any node a in a given component, $\text{search}(a)$ yields the same result, namely the bulleted node at the root of the component. The task is thus reduced to identifying the component in which node a lies.



■ **Figure 3** A tree $\mathcal{T}$ in which the top nodes of free vertices are marked as thick bullets.

Every addition of an edge to M means that two vertices are no longer free, so two bullets disappear, and the corresponding components are merged with their parents. This calls for a UNION-FIND data structure. Recall that this is a data structure that maintains a partition of a ground set, which, in our case, is $V(\mathcal{T})$. Each set has a *representative*, and the operation $\text{FIND}(a)$ (for any node a) retrieves the representative of the set containing a . Initially, the partition contains for every node a a singleton set $\{a\}$ whose representative is a . The operation $\text{UNION}(a \leftarrow c)$ forms the union of the set A containing a and the set C containing c ; the new common representative is the one of set A . We will only perform this when we know that $\text{FIND}(a) \neq \text{FIND}(c)$.

We will use the UNION-FIND data structure to perform operation *search* efficiently. All our unions will be of the form $\text{UNION}(\text{parent}(a) \leftarrow a)$, where $\text{parent}(a)$ denotes the parent of a . Recall that we assumed that the root is a top node, so $\text{parent}(a)$ is well-defined when a is not a top node.

We will maintain the following invariants.

- (M1) As long as the root of $\mathcal{T}$ has not yet been processed, the set of nodes a for which the operation $\text{UNION}(\text{parent}(a) \leftarrow a)$ has been performed is $V(\mathcal{T}) \setminus \{t(z) : z \in V(G), z \text{ is free}\}$.
- (M2) As long as the root of $\mathcal{T}$ has not yet been processed, $\text{FIND}(c) = \text{search}(c)$ for all nodes c .

Our algorithm stores no significant helper information other than the UNION-FIND data structure and is just a few lines long. (For reference, Algorithm 8 in Appendix A.2 gives the complete pseudocode.)

Initialization. Initialize $M := \emptyset$.

Initialize a UNION-FIND data structure where every node of $\mathcal{T}$ forms a singleton set.

For every node a of $\mathcal{T}$ that is not a top node, perform $\text{UNION}(\text{parent}(a) \leftarrow a)$.

Processing node a . We assume that all strict descendants of a have already been processed. If $a = t(z)$ for some z :

 Compute $c := \text{FIND}(b(z))$

 If $c \neq a$:

- Let y be the vertex with $t(y) = c$, and add edge yz to M .
- $\text{UNION}(\text{parent}(c) \leftarrow c)$.
- If a is not the root of $\mathcal{T}$ then perform $\text{UNION}(\text{parent}(a) \leftarrow a)$.

After processing the root node, we return M as the maximum matching.

Correctness. Invariant (M1) is obviously satisfied: Initially, all vertices are free, and the initialization step performs the unions for all nodes that are not top nodes. Whenever a vertex becomes matched, the union operations are performed for the corresponding top node. The only exception is that we do not apply the $\text{UNION}(\text{parent}(a) \leftarrow a)$ when a is the root of $\mathcal{T}$ (for obvious reasons). But this happens only in the last iteration, when the root is processed, an exception that is provided for in Invariant (M1).

To derive Invariant (M2) from (M1), we use the following statement:

► **Proposition 6.** *Let $\mathcal{T}$ be a tree rooted at node ρ . Let $F \subseteq V(\mathcal{T})$ be a set of nodes that contains the root ρ , and let $U := V(\mathcal{T}) \setminus F$ be its complement. Assume we have initialized a UNION-FIND data structure with each $a \in V(\mathcal{T})$ as a singleton set. After performing $\text{UNION}(\text{parent}(a) \leftarrow a)$ for all nodes $a \in U$ in any order, $\text{FIND}(c)$ will report for any node c the first node in F on the path from c to the root ρ .*

Proof. See Figure 3 for an illustration. The operation $\text{UNION}(\text{parent}(a) \leftarrow a)$ unites the set containing a with the set containing its parent. It follows by induction on $|U|$ that the sets are the connected components of the subgraph of $\mathcal{T}$ consisting of the edges from the nodes of U to their parents. Also (again by induction) each component forms a rooted tree, whose root is the representative of the set. By construction the set of roots is F .

Thus $\text{FIND}(c)$ can be described as walking upwards in the tree to the root of the component, which equals the first node in F on the path to the root of the tree. ◀

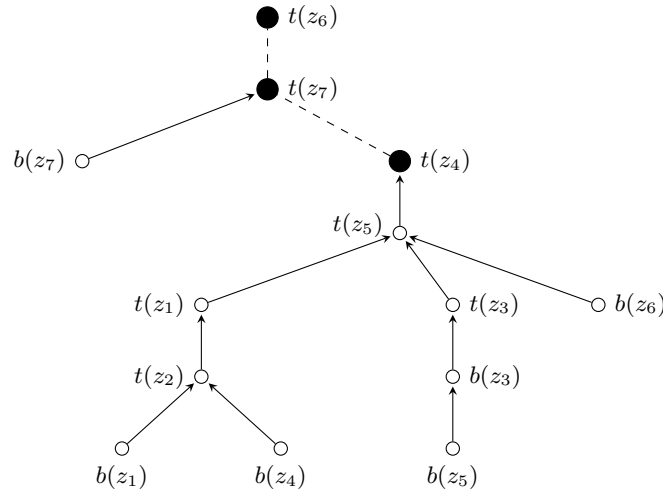
Since $\text{search}(c)$ is precisely the operation of reporting the first node in F on the path from c to the root ρ , Invariant (M2) follows as long as the root is free. This implies that our algorithm creates the same matching as Algorithm 4. Invariant (M2) does not apply after the root $a = \rho$ is processed, but then the algorithm is about to terminate anyway, and there will be no further FIND operations.

Runtime. Every UNION operation is between a node and its parent in the tree $\mathcal{T}$. This is exactly the *static tree set union* scenario, for which there is a tailor-made data structure [9]: In this situation, on a random-access machine with unit cost measure and $O(\log n)$ word length, for a universe of size N , a sequence of $N - 1$ UNION operations and k FIND operations can be carried out in $O(k + N)$ time and $O(N)$ space. (The algorithm precomputes the answers to FIND operations on small sets, and then uses table look-up on small sets combined with a conventional UNION-FIND algorithm on a universe of size $o(N)$.) We perform at most two FIND operations for every node. With this, the UNION-FIND operations take $O(|\mathcal{T}|)$ time in total, and the rest of the algorithm can be implemented with runtime $O(|\mathcal{T}| + n)$ as well.

Since the greedy algorithm is known to find a maximum matching in an RDV graph [6], we have our result for MATCHING; see Appendix B.1 for a self-contained proof.

► **Theorem 7.** *Let G be an n -vertex RDV graph with a given RDV representation of size $O(n)$. Then we can find a maximum matching of G in $O(n)$ time.* ◀

Example. On the running example of Figure 2, the top nodes are processed in the order $t(z_2)$, $t(z_1)$, $t(z_3)$, $t(z_5)$, $t(z_4)$, $t(z_7)$, $t(z_6)$. At $t(z_2)$, we have $\text{FIND}(b(z_2)) = t(z_2)$, since z_2 is free and its path consists of a single node; so nothing happens. At $t(z_1)$, $\text{FIND}(b(z_1)) = t(z_2)$, so the edge z_2z_1 is added to M , and $t(z_2)$ and $t(z_1)$ are united with their parents. At $t(z_3)$, nothing happens, and at $t(z_5)$, $\text{FIND}(b(z_5)) = t(z_3)$, so the edge z_3z_5 is added. At $t(z_4)$ and $t(z_7)$, the search returns the node itself: the neighbors z_1 , z_2 , z_5 of z_4 whose top nodes lie below $t(z_4)$ are already matched, and z_7 has no neighbor whose top node is lower than $t(z_7)$. Figure 4 shows the components of the UNION-FIND data structure at this point, before the root $t(z_6)$ is processed. Finally, $\text{FIND}(b(z_6)) = t(z_4)$, and the edge z_4z_6 is added. The resulting matching $\{z_2z_1, z_3z_5, z_4z_6\}$ is maximum, since G has only seven vertices.



■ **Figure 4** The UNION-FIND components for the running example of Figure 2 just before the root $t(z_6)$ is processed by the algorithm for MATCHING. The free vertices are z_4 , z_6 , and z_7 ; their top nodes are drawn as bullets, and the edges from these nodes to their parents (dashed) have not been united. $\text{FIND}(b(z_6))$ returns $t(z_4)$, the root of the component containing $b(z_6)$.

7 Outlook

An obvious question is whether we really need the UNION-FIND data structure for MATCHING. (This is the most complicated part of our algorithm, especially if we want runtime $O(n)$ rather than $O(n\alpha(n))$.) Can we achieve $O(n)$ runtime while storing only elementary data at each node?

Another open problem concerns transferring the results to other graph classes that have intersection representations with restricted types of objects. As mentioned, $O(n \log \log n)$ -time algorithms are known for MATCHING in circular arc graphs [16] and permutation graphs [20]. These graph classes are unrelated to RDV graphs, and they are not even chordal. Can techniques like ours be used to reduce the runtime in these graph classes to $O(n)$ as well, or at least to avoid the use of van Emde Boas trees?

The AI assistant Claude (Anthropic) was used during the preparation of this paper for proofreading (consistency of notation and terminology, typos, wording, and style), for suggestions on the organization of the presentation, and for drafting the worked examples, including Figures 2 and 4 and Table 1; the values in the examples were checked by running implementations of the algorithms. The algorithms, proofs, and results are the work of the authors, who have checked all suggested changes and take full responsibility for the content of the paper.

References

- 1 T. Biedl and P. Gokhale. Finding maximum matchings in RDV graphs efficiently. In Rahnuma Islam Nishat, editor, *Canadian Conference on Computational Geometry (CCCG 2024)*, pages 309–316, 2024.
- 2 T. Biedl and P. Gokhale. Using ray-shooting queries for sublinear algorithms for dominating sets in RDV graphs. In Jakub Kozik and Alexander Wolff, editors, *SOFSEM 2026: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, pages 216–230. Springer, 2026. doi:10.1007/978-3-032-17801-5_16.
- 3 Therese Biedl and Prashant Gokhale. Finding maximum matchings in RDV graphs efficiently. *Computing in Geometry and Topology*, 5(2):4:1–4:14, 2026. Special issue for CCCG’24. doi:10.57717/cgt.v5i2.72.
- 4 K. Booth and J.H. Johnson. Dominating sets in chordal graphs. *SIAM Journal on Computing*, 11(1):191–199, 1982. doi:10.1137/0211015.
- 5 S. W. Cheng, M. Kaminski, and S. Zaks. Minimum dominating sets of intervals on lines. *Algorithmica*, 20(3):294–308, Mar 1998. doi:10.1007/PL00009197.
- 6 E. Dahlhaus and M. Karpinski. Matching and multidimensional matching in chordal and strongly chordal graphs. *Discrete Applied Mathematics*, 84(1):79–91, 1998. doi:10.1016/S0166-218X(98)00006-7.
- 7 R. Diestel. *Graph Theory*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.
- 8 M. Farber. Domination, independent domination, and duality in strongly chordal graphs. *Discrete Appl. Math.*, 7(2):115–130, 1984. doi:10.1016/0166-218X(84)90061-1.
- 9 H. Gabow and R. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.
- 10 F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1(2):180–187, 1972. doi:10.1137/0201013.
- 11 F. Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discret. Math.*, 13(3):237–249, 1975. doi:10.1016/0012-365X(75)90021-7.
- 12 Y. Giyora and H. Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5(3):28:1–28:51, 2009. doi:10.1145/1541885.1541889.

- 13 P. Gokhale. Efficient algorithms for RDV graphs. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2025. URL: <https://hdl.handle.net/10012/21770>.
- 14 M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
- 15 U. I. Gupta, D. T. Lee, and J. Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982. doi:<https://doi.org/10.1002/net.3230120410>.
- 16 Y. Liang and C. Rhee. Finding a maximum matching in a circular-arc graph. *Information Processing Letters*, 45(4):185–190, 1993. doi:10.1016/0020-0190(93)90117-R.
- 17 E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. doi:10.1137/0214021.
- 18 A. Moitra and R. Johnson. A parallel algorithm for maximum matching on interval graphs. In *Proceedings of the International Conference on Parallel Processing, ICPP '89*, pages 114–120. Penn State University Press, 1989.
- 19 C. Monma and V. Wei. Intersection graphs of paths in a tree. *J. Comb. Theory, Ser. B*, 41(2):141–181, 1986. doi:10.1016/0095-8956(86)90042-0.
- 20 C. Rhee and Y. Liang. Finding a maximum matching in a permutation graph. *Acta Informatica*, 32(8):779–792, 1995. doi:10.1007/BF01178659.

A

 Streamlined Pseudocode for our Three Algorithms

A.1 Streamlined Algorithm for INDEPENDENTSET

Algorithm 5 Streamlined algorithm for INDEPENDENTSET

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

- 1 initialize $I := \emptyset$, set $state(z) := \text{free}$ for all $z \in V(G)$, $L(a) := \emptyset$ for all $a \in V(\mathcal{T})$
- 2 **for** $a \in V(\mathcal{T})$, in leaf-to-root order **do**
- 3 **for all children** c **of** a **do** append $L(c)$ to $L(a)$ (this empties $L(c)$)
- 4 **if** $a = b(z)$ **for some** $z \in V(G)$ **then** add z to $L(a)$
- 5 **if** $a = t(z)$ **for some** $z \in V(G)$ **and** $state(z) = \text{free}$ **then**
- 6 add z to I
- 7 **for all** $y \in L(a)$ **do** delete y from $L(a)$ and set $state(y) := \text{blocked}$

Result: I

A.1.1 A Numeric Abstraction for INDEPENDENTSET

We can reduce the algorithm even more: Instead of storing a list $L(a)$ we store only the lowest top node of a vertex in $L(a)$:

$$\begin{aligned} \ell(a) &:= \max \{ \text{depth}(t(z)) : z \in L(a) \} \\ &= \max \{ \text{depth}(t(z)) : z \in \beta(a) \text{ and } z \text{ is free} \} \end{aligned}$$

For any empty list, we take the maximum to be the sentinel value -1 (which has the same effect as the logically sounder value $-\infty$). The value $\ell(a)$ allows us to find out when a vertex should be added to I . The reason is that all vertices $z \in L(a) \subseteq \beta(a)$ have $\text{depth}(t(z)) \leq \text{depth}(a)$, by definition. Thus, if $\ell(a) = \text{depth}(a)$, it means that there must be some **free** vertex $z \in L(a)$ with $\text{depth}(t(z)) = \text{depth}(a)$, and hence $t(z) = a$.

Processing node a . We assume that all strict descendants of a have already been processed.

1. (Accumulate the data from the children.) Set $\ell(a) := \max \{ \ell(c) : c \text{ a child of } a \}$.
2. (Update if a path starts.) If a is a bottom node, say $a = b(z)$:
 - Set $\ell(a) := \max\{\ell(a), \text{depth}(t(z))\}$.
3. (Add to solution.) If $\ell(a) = \text{depth}(a)$, then a must be a top node, say $a = t(z)$.
 - Add z to I . // z was **free** until now.
 - Set $\ell(a) := -1$. // Empty the list $L(a)$.

Algorithm 6 summarizes the complete algorithm in streamlined form.

■ **Algorithm 6** An even more streamlined algorithm for INDEPENDENTSET

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

```

1 initialize  $I := \emptyset$ 
2 for  $a \in V(\mathcal{T})$ , in leaf-to-root order do
3   set  $\ell(a) := \max(\{ \ell(c) : c \text{ a child of } a \} \cup \{-1\})$ 
4   if  $a = b(z)$  for some  $z \in V(G)$  then set  $\ell(a) := \max\{\ell(a), \text{depth}(t(z))\}$ 
5   if  $a = t(z)$  for some  $z \in V(G)$  and  $\ell(a) = \text{depth}(a)$  then
6     add  $z$  to  $I$ 
7     set  $\ell(a) := -1$  // Empty the list  $L(a)$ .

```

Result: I

A.2 Streamlined Algorithm for DOMINATINGSET and MATCHING

Algorithm 7 shows a streamlined version of the algorithm for DOMINATINGSET from Section 5, which uses a global set D .

■ **Algorithm 7** Streamlined algorithm for DOMINATINGSET

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

```

1 initialize  $D := \emptyset$ 
2 for  $a \in V(\mathcal{T})$ , in leaf-to-root order do
3   set  $h(a) := \min(\{ h(c) : c \text{ a child of } a \} \cup \{\infty\})$ 
4   if  $h(a) < \infty$  then
5     let  $c^*$  be the child of  $a$  with  $h(c^*) = h(a)$ , and set  $z_{\text{high}}(a) := z_{\text{high}}(c^*)$ 
6   set  $d(a) := \min(\{ d(c) : c \text{ a child of } a \} \cup \{\infty\})$ 
7   set  $u(a) := \max(\{ u(c) : c \text{ a child of } a \} \cup \{-\infty\})$ 
8   if  $d(a) < \infty$  then set  $u(a) := -\infty$ 
9   if  $a = b(z)$  for some  $z \in V(G)$  then
10     if  $\text{depth}(t(z)) < h(a)$  then update  $z_{\text{high}}(a) := z$ ,  $h(a) := \text{depth}(t(z))$ 
11     if  $d(a) = \infty$  then update  $u(a) := \max\{u(a), \text{depth}(t(z))\}$ 
12   if  $a = t(z)$  for some  $z \in V(G)$  then
13     if  $u(a) = \text{depth}(a)$  then
14       add  $z_{\text{high}}(a)$  to  $D$ , set  $d(a) := h(a)$ , set  $u(a) := -\infty$ 
15     if  $h(a) = \text{depth}(a)$  then set  $h(a) := \infty$ 
16     if  $d(a) = \text{depth}(a)$  then set  $d(a) := \infty$ 

```

Result: D

Algorithm 8 shows the algorithm for MATCHING from Section 6 in terms of UNION-FIND operations.

■ **Algorithm 8** Streamlined algorithm for MATCHING in terms of UNION-FIND operations

Input: RDV graph G with RDV representation $(\mathcal{T}, t, b)$

```

1 initialize the matching  $M := \emptyset$ 
2 initialize a UNION-FIND data structure where every node of  $\mathcal{T}$  forms a singleton set
3 for every node  $a$  of  $\mathcal{T}$  that is not a top node do UNION(parent( $a$ )  $\leftarrow$   $a$ )
4 for each node  $a \in V(\mathcal{T})$ , in leaf-to-root order do
5   if  $a = t(z)$  for some vertex  $z$  then
6     compute  $c := \text{FIND}(b(z))$ 
7     if  $c \neq a$  then
8       let  $y$  be the vertex with  $t(y) = c$ 
9       add edge  $yz$  to  $M$ 
10      UNION(parent( $c$ )  $\leftarrow$   $c$ )
11      if  $a$  is not the root of  $\mathcal{T}$  then UNION (parent( $a$ )  $\leftarrow$   $a$ )

```

Result: M

B Equivalence of Algorithms 2 and 3

In this section, we argue that the greedy algorithm for MATCHING (Algorithm 2) produces exactly the same output as the delayed greedy algorithm (Algorithm 3). Both algorithms were stated with a vertex ordering specifically designed for RDV graphs, but they can in fact be applied to *any* vertex ordering (as long as optimality of the result is not required), and we argue equivalence even in this situation.

► **Lemma 8.** *For any graph G (not necessarily an RDV graph) and any fixed vertex ordering $z_1, \dots, z_n$, Algorithm 3 produces the same matching as Algorithm 2 when both are applied with this ordering.*

Proof. The proof is by induction on the number of vertices. If z_1 is an isolated vertex, it is left unmatched by both algorithms, and the removal of z_1 does not change anything for either algorithm.

Otherwise, we first argue that z_1 is matched to the same vertex z_k by both algorithms: Initially, all vertices are free. Suppose that Algorithm 2 matches z_1 to z_k because z_k is the lowest-numbered neighbor of z_1 . Then, when Algorithm 3 successively considers the vertices $z_q = z_2, z_3, \dots, z_{k-1}$, none of these vertices is adjacent to z_1 ; hence z_1 is still free when $z_q = z_k$ is considered, and the algorithm will of course choose the edge $z_1 z_k$.

We then observe that the removal of z_1 and z_k does not affect the decisions about the remaining vertices. This is obvious for Algorithm 2, because z_1 and z_k are matched in the first round, and then the vertices z_1 and z_k are no longer free and out of the game.

In Algorithm 3, none of the vertices $z_q = z_2, z_3, \dots, z_{k-1}$ is adjacent to z_1 , and hence the absence of z_1 is not noticed. Neighbors z_i with $i > q$ such as $z_i = z_k$ do not play a role. Later, when the vertices $z_q = z_{k+1}, z_{k+2}, \dots$ are considered, z_1 and z_k are no longer free in the first place, and hence their removal makes no difference.

Thus, we can remove z_1 and z_k (or only z_1 , in case z_1 is isolated) from the input without changing the course of the algorithms on the remaining vertices. Thus, by induction, the remaining matching edges are also the same. ◀

B.1 Optimality Proof for Algorithms 2 and 3

Extending the arguments in the proof of Lemma 8, optimality for Algorithms 2 and 3 can be established by an easy exchange argument. For convenience, we include this self-contained proof here. In contrast to Lemma 8, we must now of course insist that the ordering $z_1, \dots, z_n$ places z_i before z_j whenever $t(z_i)$ is a child of $t(z_j)$.

As mentioned in the main text, optimality of the greedy algorithm is known for a strong perfect elimination order in a strongly chordal graph, due to Dahlhaus and Karpinski [6]. For RDV graphs, it is not hard to show that such a strong perfect elimination order is obtained with the leaf-to-root order of top nodes that we assumed in Algorithms 2 and 3. (For the reverse level order of $\mathcal{T}$, this has been shown in [3, 13].) However, instead of deriving optimality from the results on strongly chordal graphs [6], our proof considers RDV graphs directly.

► **Theorem 9.** *The greedy Algorithms 2 and 3 compute a maximum matching.*

Proof. The argument is based on the following claim:

▷ **Claim 10.** If the algorithm matches z_1 with z_q then there is a maximum matching that matches z_1 with z_q .

From this, optimality follows by induction on n , along the lines of the proof of Lemma 8: If z_1 is matched with z_q , then the algorithm, when restricting our attention to the remaining vertices, behaves as if z_1 and z_q were never present, and thus, by induction, it computes a maximum matching on $G \setminus \{z_1, z_q\}$. By the claim, that matching together with the edge $z_1 z_q$ is a maximum matching for G .

If z_1 is an isolated vertex, then again, the algorithm behaves as if z_1 were never present, and the argument is analogous. ◀

Proof of Claim 10. Let $w_1, w_2, \dots, w_k$ be the neighbors of z_1 , in the order in which they appear in the list $z_1, \dots, z_n$. Thus, the algorithm matches z_1 with $w_1 = z_q$.

We apply Observation 2 to the vertices w_i : Let $a = t(z_1)$. Then either $w_i \in \beta(a)$, or its top node $t(w_i)$ is a strict descendant of a . However, there are no top nodes that are strict descendants of a , because the corresponding vertices would have to come before z_1 .

We conclude that all neighbors w_i are in $\beta(a)$. Their top nodes lie on the path from a to the root. This has an important consequence: When checking adjacency with w_i , only the path from a to $t(w_i)$ matters, and the path below a can be ignored. The reason is that there are no top nodes below a .

By our assumption on the ordering, the top nodes $t(w_1), t(w_2), \dots, t(w_k)$ appear in this order on the path from a to the root, and consequently, the paths from a to $t(w_i)$ (which determine the neighbors of w_i) are nested. From this we conclude that the neighborhoods are nested as well: $N[w_1] \subseteq N[w_2] \subseteq \dots \subseteq N[w_k]$. Intuitively, this means that the best matching partner to choose greedily for z_1 is w_1 , because all other w_i have more neighbors and thus more possibilities to be matched in the remaining graph.

Formally, we prove the claim as follows. Let M^* be a maximum matching. Since it is a maximum matching, it cannot leave both z_1 and w_1 unmatched.

If M^* matches only one of the vertices z_1 and w_1 , we can always include the edge $z_1 w_1$, at the expense of removing the matching edge incident to z_1 or w_1 , and we get a matching that is no smaller than before.

Finally, suppose that M^* matches z_1 with some vertex w_i with $i > 1$, and w_1 is matched with some vertex $v \in N[w_1]$. Then we can replace these two edges by the edges $z_1 w_1$ and $w_i v$. The edge $w_i v$ must exist, since $v \in N[w_1] \subseteq N[w_i]$. ◀