

Efficient Algorithms for RDV graphs

by

Prashant Gokhale

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Prashant Gokhale 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Statement on originality of work: Chapter 3 and most of Chapter 5 were written for a paper that was published at CCCG 2024 [4]. These chapters were co-authored with Prof. Therese Biedl.

Prashant Gokhale is the sole author of Chapters 1, 2, and 4. These chapters were written under the supervision of Prof. Therese Biedl.

Statement on AI use: ChatGPT¹, a large language model, was used in certain parts of this thesis. Its use was strictly limited to (i) improving the clarity of the English text and (ii) fine-tuning TikZ code for figure generation. In particular, it was not used for content creation. This usage was discussed in advance with the supervisor in accordance with the University’s Generative AI policy².

¹OpenAI (2025), ChatGPT (Feb 2025 version) [Large Language Model], <https://openai.com>

²<https://uwaterloo.ca/current-graduate-students/academics>

Abstract

In this thesis, we study the maximum matching and minimum dominating set problem in RDV graphs, i.e., graphs that are vertex-intersection graphs of downward paths in a rooted tree. A straightforward implementation of these algorithms would require $O(n+m)$ time. We improve their efficiency by transforming the question about the neighborhood of v into a type of range query amid a set of horizontal and vertical line segments. Our algorithms run in $O(n \log n)$ time, presuming a $O(n)$ -sized intersection representation of the graph is given. In addition, our techniques can also be used to obtain faster algorithms for maximum independent set and perfect k -clique packing in RDV graphs.

Acknowledgements

I am deeply grateful to my thesis supervisor, Prof. Therese Biedl, for her exceptional supervision, guidance, and flexibility, especially as my research interests evolved. I also want to thank Prof. Ian Munro and Prof. Sophie Spirkel for serving as readers on my thesis committee.

Table of Contents

Author’s Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
1 Introduction	1
2 Preliminaries: RDV graphs and friends	8
2.1 Notation	8
2.2 Intersection graphs of subtrees in a tree	9
2.2.1 Chordal Graphs	10
2.2.2 Strongly Chordal Graphs	12
2.2.3 Path Graphs	13
2.2.4 RDV graphs	13
2.2.5 Interval Graphs	16
2.3 Tools for RDV graphs	17
2.4 Data Structures from Computational Geometry	19

2.4.1	Priority Search Trees	19
2.4.2	Dynamic Orthogonal Ray Shooting	19
3	Maximum Matching	21
3.1	Introduction	21
3.2	Greedy Strategy for matching	23
3.3	Optimality for strongly chordal graphs	24
3.4	Neighborhood queries in RDV graphs	26
3.5	Sublinear $O(n \log n)$ matching algorithm	30
4	Minimum Dominating Set	33
4.1	Introduction	33
4.2	Greedy strategy	34
4.3	Optimality for strongly chordal graphs	35
4.4	Sublinear $O(n \log n)$ algorithm	37
4.4.1	Implementing Operation A in $O(\log n)$ time per iteration	38
4.4.2	Implementing Operation B in $O(\log n)$ time per iteration	38
4.4.3	Putting everything together, and outlook	40
5	Conclusion	43
5.1	Perfect Clique Packing in RDV graphs	44
5.2	Maximum Independent Set in RDV graphs	45
5.3	Other graph classes	47
	References	50
	Appendix	55
A	Supplementary Code	56

Chapter 1

Introduction

A *graph* $G = (V, E)$ is a discrete mathematical structure that represents pairwise relationships (edges) between vertices, where V denotes the vertices, and E the edges (throughout this thesis, we disallow edges from a vertex to itself). Graphs are ubiquitous in computer science and mathematics, and arise in many real-world applications. *Graph algorithms* is a subfield of theoretical computer science which focuses on designing efficient algorithms on graphs.

In this thesis, we design graph algorithms that are specifically tailored for a certain class of graphs—RDV graphs—that can be defined via geometric representations. These algorithms will be significantly more efficient than algorithms for general graphs. The rest of this chapter will give definitions and motivations for the graph classes that we study.

An *interval graph* is a graph constructed from a set of intervals on a line, where each interval corresponds to a vertex, and an edge exists between two vertices if their intervals overlap. Conversely, a graph G has an *interval representation* if there exists a set of intervals whose interval graph is G . Refer to Figure 1.1 for an example of an interval graph. Interval graphs are commonly used to model scheduling problems, and many algorithmic problems admit more efficient solutions on interval graphs than on general graphs. For instance, finding the largest clique (a subset of vertices where every pair is adjacent) is **NP-Hard** in general [27] but can be solved in linear time on interval graphs [36]. Interval graphs have been the subject of extensive research, and stating all related results is beyond the scope of this discussion. However, the interested reader can consult Chapter 8 of Golumbic’s book [33] for a more in-depth discussion of interval graphs.

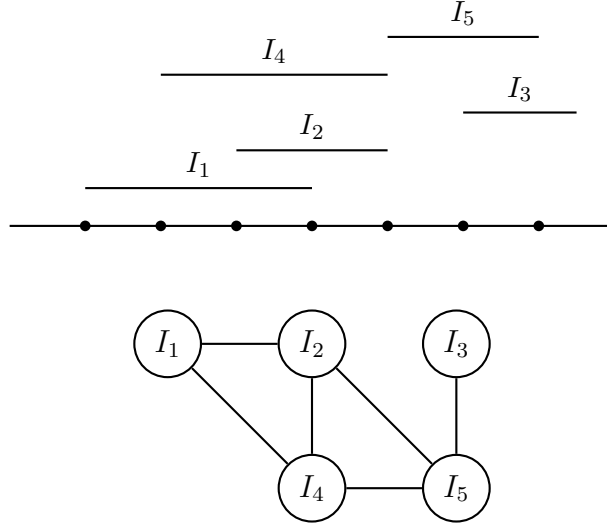


Figure 1.1: An interval graph.

Interval graphs are a specific instance of a broader class of graphs known as *intersection graphs* [47]. The *intersection graph* of a family $\mathcal{F}$ of sets is a graph G with a vertex for every set $S \in \mathcal{F}$ and an edge between two vertices S_1, S_2 if and only if $S_1 \cap S_2 \neq \emptyset$. A graph G is said to have an *intersection representation* $\mathcal{F}$ if the intersection graph of $\mathcal{F}$ is G . Every graph G admits a trivial intersection representation: for each vertex v , include in $\mathcal{F}$ the set of all edges incident to v . So, the focus is typically on intersection representations with specific constraints. For example, in interval graphs, all sets in the representation are restricted to be intervals on a line.

Intersection graphs are widely studied in mathematics for their structural properties, and in computer science from an algorithmic perspective. For example, all interval graphs are *perfect graphs*—graphs in which the chromatic number (the minimum number of colors required to color the vertices so that no two adjacent vertices share the same color) equals the size of the largest clique, both in the graph itself and in all of its induced subgraphs. Perfect graphs have been extensively studied in the literature. For further details, the interested reader can consult Golumbic’s book [33] and Trotignon’s survey article [62]. Additionally, interval graphs are *chordal*—graphs in which every induced cycle is a triangle, or equivalently, every cycle with more than three vertices has a *chord* (an edge joining two

non-consecutive vertices). However, chordal graphs form a much broader classes than interval graphs, and many problems (e.g., the dominating set problem) are hard to solve on them. Therefore, we will focus on graph classes that lie between interval graphs and these broader classes.

There are several ways to define such generalizations of interval graphs. This thesis focuses on a specific one known as *RDV graphs*, which we informally describe here (for further details, see Chapter 2). Interval graphs can be viewed as intersection graphs of path subgraphs within a host path graph, where the path subgraphs represent intervals, and the host path corresponds to a straight line. Generalizing this concept, RDV graphs are defined as intersection graphs of downward path subgraphs within a rooted host tree. RDV graphs are also a subclass of both perfect graphs and chordal graphs. Their name is derived from *vertex-intersection paths of downward paths in a rooted tree* [52]. Refer to Figure 1.2 for an example of an RDV graph.

RDV graphs were first introduced by Renz in 1970 [59] and later studied by Gavril in 1974 [28]. Dietz, Furst, and Hopcroft examined them in 1979 [21] while studying the consecutive retrieval problem. In this problem, given n records and a collection of subsets of these records, the goal is to find a linear ordering of the records such that each subset appears as a consecutive interval in the ordering. This can be framed as an interval graph recognition problem. Relaxing the ordering condition from a line to a tree transforms the problem to an RDV graph recognition problem, where the task is to determine if a given graph has an RDV representation. Dietz, Furst and Hopcroft [21] developed a linear time algorithm for this *generalized* consecutive retrieval problem. Later, Swaminathan and Wagner [61] presented a simpler algorithm, though at the cost of an additional inverse Ackermann factor in the running time. Moreover, both of these recognition algorithms also yield an intersection representation. The term “RDV graphs” was introduced by Monma and Wei [52] in 1986, who studied RDV graphs and related classes of intersection graphs.

RDV graphs are also interesting from a structural graph theory perspective. Recall that all interval graphs are *chordal* graphs. Can chordal graphs be characterized by an intersection representation? Indeed, they can—chordal graphs are precisely the intersection graphs of *subtrees of a host tree* (formally defined in Chapter 2), a result first proven by Gavril [28]. This immediately places RDV graphs as an intermediate class between interval graphs and chordal graphs. See Chapter 2 for a more detailed discussion on how RDV graphs relate to these and other closely associated graph classes.

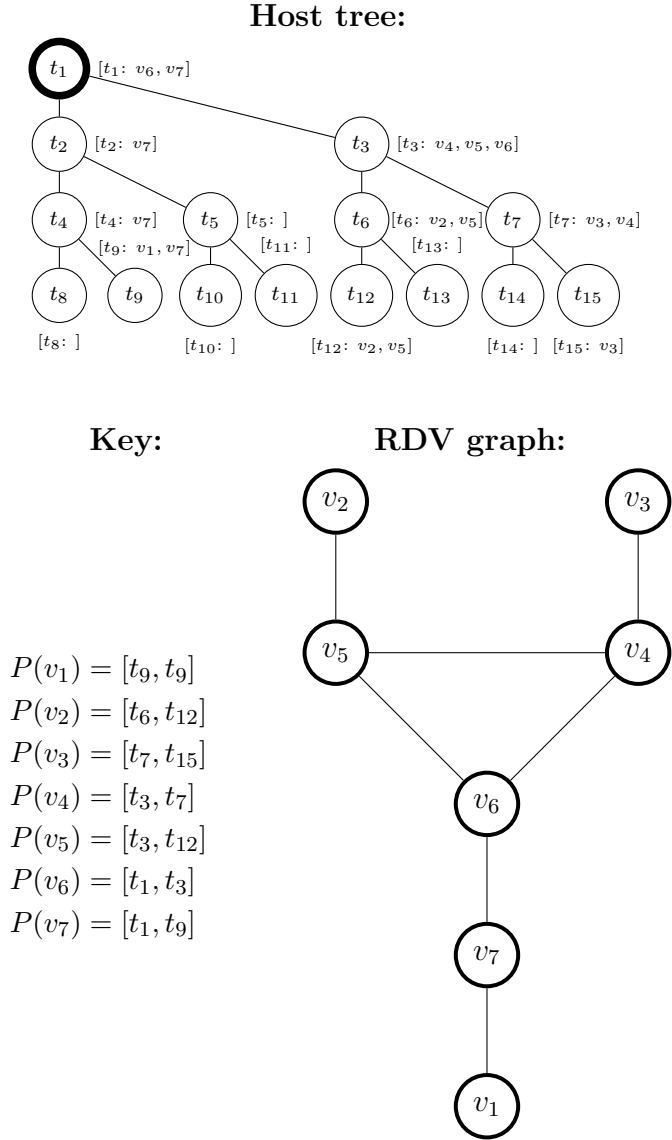


Figure 1.2: An example of an RDV graph (labelled with v_i) and its rooted host tree (labelled with t_i). The downward path corresponding to vertex v_i is $P(v_i)$, and is indicated by the endpoint nodes of that path. Next to each node of the host tree, the paths that include it have been listed.

The main goal of this thesis can be briefly summarized as follows: For interval graphs, many algorithms benefit from exploiting the geometric nature of their intersection repre-

sentation to improve efficiency. Can this approach be applied to RDV graphs as well? We show that the answer is yes, providing new results for two well-known and widely studied graph problems: (i) finding a maximum matching and (ii) finding a minimum dominating set. In particular, we present $O(n \log n)$ algorithms for both of these problems. Note that RDV graphs may well have $m \in \omega(n \log n)$ edges, so our algorithms are sublinear in the graph “size” $n + m$. In particular, a sublinear runtime means that the algorithm is not even permitted to read all the edges. Instead, we leverage the intersection representation (which is a way of compactly representing a graph) to achieve an $O(n \log n)$ runtime, given that a suitable intersection representation is provided as input. It is important to note that our algorithm does not require access to the underlying graph directly; it only utilizes the intersection representation. This is significant because the graph itself may require $\Omega(n^2)$ space for storage, whereas the intersection representation only needs $O(n)$ space.

Later, we also briefly show how our techniques can generalize to other problems as well. Specifically, we achieve similar speed-ups for maximum independent set and perfect k -clique packing as well.

At this point, one might ask: Why study RDV graphs at all? While interval graphs frequently arise in practical scenarios, analogous real-world examples for RDV graphs are harder to pinpoint, though the generalized retrieval problem provides some motivation. The strongest justification for studying RDV graphs, however, is structural. They occupy a position between two extensively studied graph classes: interval graphs and chordal graphs. In many ways, RDV graphs appear to be “close” to interval graphs. Efficient algorithms have been developed for various problems on RDV graphs, including graph isomorphism [1] and counting maximal independent sets [42], among many others. However, in slightly larger graph classes, such as path graphs (see Chapter 2), they become computationally harder or intractable [1, 42, 43, 58].

In this spirit, our work shows that maximum matching and minimum dominating set can be computed in sublinear time on RDV graphs, just as on interval graphs. Looking ahead, it would be interesting to explore the limits of our approach. For example, is it possible to design sublinear algorithms for matching using the intersection representation of chordal graphs (or a smaller superclass of RDV graphs, such as the strongly chordal graphs defined in Chapter 2)? To the best of our knowledge, there are no deterministic exact algorithms for chordal graphs that have a better runtime than the ones for general graphs.

We conclude this chapter by outlining our techniques. Chordal graphs are also defined by the existence of a vertex order called a *perfect elimination order*, where, for any vertex v , the neighbors of v that appear later in the order form a clique. Many algorithms for chordal graphs (and their subclasses) exploit properties of the perfect elimination order. For instance, one can efficiently list all maximal cliques in a chordal graph by considering each vertex v along with all its successors and testing whether the resulting clique is maximal. For an interval graph, obtaining a perfect elimination order is straightforward: simply order the intervals by their right endpoints in increasing order. This order also forms the basis of the classic greedy scheduling algorithm for interval graphs. But what about RDV graphs? Interestingly, a perfect elimination order can also be obtained in $O(n)$ time for RDV graphs by listing the vertices (paths) in the order of *decreasing* depth of their top nodes.

For RDV graphs, previously known algorithms for maximum matching and minimum dominating set make use of this order [19, 6]. These algorithms are conceptually “greedy”, as they process vertices sequentially in the perfect elimination order, and greedily match the current vertex v or augment the current dominating set according to a rule that depends on the neighborhood $N[v]$ of v . While a straightforward implementation of these algorithms would require $O(\deg(v))$ time per vertex, we improve their efficiency by transforming the question about the neighborhood of v into a type of range query amid a set of horizontal and vertical line segments. Refer to Theorem 3.5.3 in Chapter 3 for details. This transformation enables us to use dynamic computational geometry data structures to efficiently answer queries such as: “Given a vertex v , find the vertex u in $N[v]$ that appears earliest or latest in the perfect elimination order” in $O(\log n)$ amortized time.

Interestingly, Theorem 3.5.3 also provides an *implicit representation* of RDV graphs, which refers to an adjacency labeling scheme using $O(\log n)$ bits. Implicit representations were introduced by Muller [53] and Kannan, Naor and Rudich [38]. In recent years, they have been studied in relation to the so-called implicit graph conjecture (see [37] for more details). While a previous implicit representation for RDV graphs was already known [38], our version allows the use of computational geometry data structures to improve the efficiency of the algorithms discussed earlier.

The thesis is organized as follows. Chapter 2 defines the RDV graph class, related graph classes, and preliminaries. Chapters 3 and 4 present $O(n \log n)$ algorithms for maximum matching and minimum dominating set in RDV graphs, respectively. Chapter 3

also contains Theorem 3.5.3, which provides a geometric method for checking adjacency relationships in RDV graphs and is essential for both the matching and dominating set algorithms. Finally, Chapter 5 concludes the thesis with several open problems and potential future research directions.

Chapter 2

Preliminaries: RDV graphs and friends

This chapter formally introduces the RDV graph class, which is the central focus of this thesis. We also define several related graph classes—chordal graphs, strongly chordal graphs, path graphs, and interval graphs—and examine their connections to RDV graphs. Lastly, we establish notation and conventions for the remainder of the thesis.

2.1 Notation

A *graph* $G = (V, E)$ is a discrete mathematical structure representing pairwise relationships (edges) between vertices, where V is the set of vertices and E is the set of edges. A graph is *simple* if it has no loop edges, meaning $(v, v) \notin E$ for all $v \in V$, and no multiple edges between the same pair of vertices. Throughout this thesis, all graphs are assumed to be simple.

A *cycle* in a graph is a sequence of vertices that forms a closed loop, where each consecutive pair is connected by an edge. A *path* in a graph is a sequence of distinct vertices such that any two consecutive vertices are joined by an edge. A graph is *connected* if there is a path from any vertex to any other vertex. A *tree* is a connected graph with n vertices that contains no cycles. Any tree with n vertices has exactly $n - 1$ edges. A tree can be *rooted* by identifying a vertex as root. This partitions the vertices into *levels*

according to their distance (shortest length of a path) from the root, also known as their *depth*. In a rooted tree, a vertex's *parents* refers to its adjacent vertex at a lower depth, while its *children* are its adjacent vertices at a higher depth. A *leaf* in a rooted tree is a vertex with no children. The *descendants* of a vertex v in a rooted tree refers to all vertices (including v) reachable from v by following a downward path from v to some leaf.

We write $A \subseteq B$ to indicate that A is a (possibly not strict) subset of B . The notation $[n]$ represents the set $\{1, 2, \dots, n\}$.

Definition 2.1.1 (Open and Closed neighborhoods of a vertex). *For a vertex v , we denote its open and closed neighborhoods by $N(v)$ and $N[v]$, respectively, where*

$$N(v) = \{u \mid (u, v) \in E\}, \quad N[v] = N(v) \cup \{v\}.$$

Definition 2.1.2 (Vertex order of a Graph). *A vertex order of a graph is a linear arrangement of its vertices, denoted by $v_1, v_2, \dots, v_n$. Given a specific vertex order, we define*

$$N_{<}(v_i) = \{v_j \mid v_j \in N(v_i), j < i\}$$

as the set of neighbors of v_i that appear earlier in the order. Similarly, we define $N_{>}(v_i)$, $N_{\leq}[v_i]$, and $N_{\geq}[v_i]$.

A *subgraph* of G is a graph formed by a subset of its edges. Given a vertex subset $H \subseteq V$, the *induced subgraph* on H , denoted by $G[H]$, contains all edges of G whose endpoints are both in H . Formally, the induced subgraph is defined as $G[H] = (H, E_H)$, where the edge set is given by

$$E_H = \{(u, v) \mid (u, v) \in E, u \in H, v \in H\}.$$

2.2 Intersection graphs of subtrees in a tree

Definition 2.2.1 (Intersection graphs and Intersection Representation). *The intersection graph of a family $\mathcal{F}$ of subsets of a universe $\mathcal{U}$ is a graph G where each set $S \in \mathcal{F}$ (where $S \subseteq \mathcal{U}$) corresponds to a vertex, and two vertices S_1 and S_2 are adjacent if and only if $S_1 \cap S_2 \neq \emptyset$. Conversely, we say that a graph G has an intersection representation $\mathcal{F}$ if G is the intersection graph of $\mathcal{F}$.*

We focus on intersection graphs where the universe is the vertex set of a tree, often called the *host tree*, and the family of subsets consists of *subtrees* of the host tree.

Definition 2.2.2 (Intersection graphs of subtrees of a host tree). *Intersection graphs of subtrees of a host tree are defined as intersection graphs where the universe $\mathcal{U}$ is the vertex set V_T of a tree $T = (V_T, E_T)$, often called the *host tree*, and the family of subsets $\mathcal{F} \subseteq 2^{V_T}$ consists of subtrees of T . That is, for each $S \in \mathcal{F}$, the induced subgraph $T[S]$ is a tree.*

Such an intersection representation is sometimes referred to as a *clique-tree* [52] or a *characteristic tree* [29]. For the rest of this thesis, we will use the term *host tree* to refer to T , and the term *tree intersection representation* to refer to T along with $\mathcal{F}$. Additionally, as a convention, we use the term *node* to refer to the vertices of the host tree, to distinguish them from the vertices of the intersection graph. The host tree can be rooted by selecting a node as the root, which also roots the subtrees. For any vertex $v \in G$, let $T(v)$ represent the corresponding subtree in the intersection representation. If the subtrees are restricted to paths, we denote them by $P(v)$ instead.

Given a rooted host tree, we define the following vertex order by sorting the subtrees by decreasing depth of their roots. We call this a *bottom-up* enumeration.

Definition 2.2.3 (Bottom-up enumeration). *Let G be a graph with a tree intersection representation with rooted host tree T . For any vertex v , let $t(v)$ be the topmost (closest to the root) vertex in the subtree $T(v)$ of v . A bottom-up enumeration of G is a vertex order obtained by sorting vertices by decreasing distance of $t(v)$ to the root, breaking ties arbitrarily.*

For example, the vertices of the graph in Figure 2.1 are enumerated according to a bottom-up enumeration.

This framework allows us to define various graph classes by imposing restrictions on the types of permitted subtrees or host tree.

2.2.1 Chordal Graphs

A *chordal graph* is a graph in which every cycle of length at least four has a *chord*, that is, an edge connecting two non-adjacent vertices in the cycle. In 1974, Gavril proved that

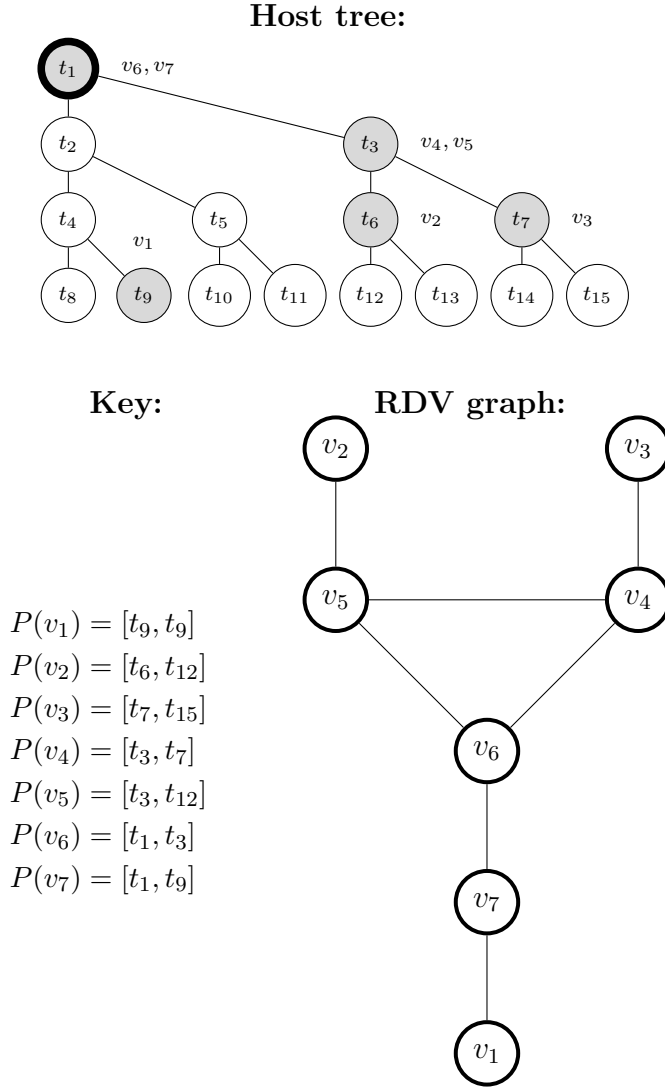


Figure 2.1: $v_1, v_2, \dots, v_7$ is a bottom-up enumeration of an RDV graph. The nodes corresponding to each $t(v_i)$ are highlighted and sorted by decreasing distance to the root.

chordal graphs are precisely the intersection graphs of subtrees in a tree [28]. Another characterization of chordal graphs is that they are those graphs which admit a perfect elimination order.

Definition 2.2.4 (perfect elimination order). *A perfect elimination order of a graph G is a vertex order $v_1, v_2, \dots, v_n$ such that $N_{\geq}[v_i]$ induces a clique for all $i \in [n]$.*

It is easy to see that a bottom-up enumeration in an intersection representation gives a perfect elimination order [28]. Several problems, such as graph coloring and maximum clique, are NP-hard on general graphs, but can be solved in polynomial time on chordal graphs. Many of these algorithms exploit the existence of a perfect elimination order. For further details on chordal graphs, see Chapter 4 of Golumbic’s book [33].

2.2.2 Strongly Chordal Graphs

Strongly chordal graphs form a subclass of chordal graphs, and were first studied by Farber in the 1980s [23]. They are defined as the class of graphs that admit a *strong perfect elimination order*.

Definition 2.2.5 (strong perfect elimination order). *A strong perfect elimination order of a graph G is a vertex order $v_1, v_2, \dots, v_n$ such that for any $i, j, k, l \in [n]$ with $i \leq j$ and $k \leq l$, if $v_k, v_l \in N[v_i]$ and $v_k \in N[v_j]$, then $v_l \in N[v_j]$.*

Observe that a strong perfect elimination order is always a perfect elimination order. Setting $i = k$ in Definition 2.2.5 ensures that for any $i < j < l$, if $v_i v_j$ and $v_i v_l$ are edges, then $v_j v_l$ must be as well. Thus, $N_{\geq}[v_i]$ induces a clique for all $i \in [n]$.

The initial motivation for studying strongly chordal graphs stemmed from the fact that while the minimum dominating set problem is NP-hard on chordal graphs [6], it admits a polynomial-time algorithm on strongly chordal graphs [25]. Strongly chordal graphs can also be characterized as intersection graphs of subtrees in a tree, with additional constraints on the allowed subtrees. For further details, see Farber’s thesis [23] and an expository article by Biedl [5].

Strongly chordal graphs form a strict subclass of chordal graphs. An example of a chordal graph that is not strongly chordal is depicted in Figure 2.2. This graph, referred to as a 4-trampoline, belongs to a broader family of graphs known as k -trampolines. A k -trampoline is a graph on $2k$ vertices, where the set $\{v_{k+1}, \dots, v_{2k}\}$ induces a k -clique. Additionally, the adjacency relations are defined as follows: v_1 is adjacent to both v_k and v_{k+1} , v_2 is adjacent to both v_{k+1} and v_{k+2} , continuing in this pattern until v_k , which is adjacent to both v_{2k} and v_{k+1} . Farber [24] showed that any k -trampoline graph for $k \geq 3$ is not strongly chordal. However, these graphs are readily seen to be chordal, as every cycle of length at least four contains a chord.

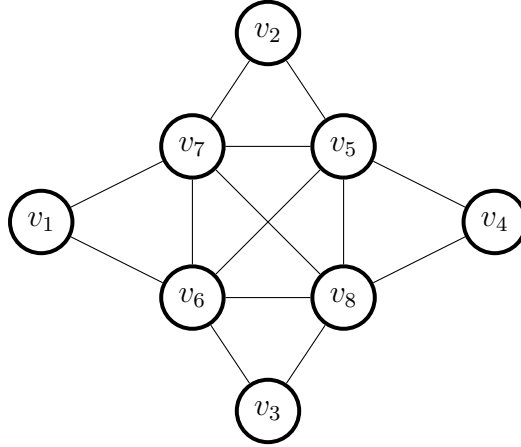


Figure 2.2: The *4-trampoline*—a graph which is chordal but not strongly chordal

2.2.3 Path Graphs

Path graphs are the intersection graphs of *paths* in a tree. In the set-up of Definition 2.2.2, this means that for each $S \in \mathcal{F}$, the induced subgraph $T[S]$ is a path. Since every path is a tree, path graphs are necessarily chordal; however, they are not always strongly chordal. In particular, the *4-trampoline* graph in Figure 2.2 is a path graph [4] (also see Figure 5.1 in Chapter 5 for another example).

Path graphs were first studied by Gavril [28]. Various subclasses of path graphs can be defined by placing additional constraints on the allowed paths. For further details, see the work of Monma and Wei [52]. In this thesis, we focus primarily on one such subclass, RDV graphs, which we define shortly.

2.2.4 RDV graphs

RDV graphs are the intersection graphs of *downward paths* in a host tree T , where a downward path is a path that monotonically increases in depth. Formally, a downward path is a sequence of vertices $p_1, p_2, \dots, p_n \subseteq V_T$ such that for each $i < n$, the vertex p_{i+1} is a child of p_i in the tree T . See Figure 1.2 for an example. We will occasionally refer to the tree intersection representation of an RDV graph as its *RDV representation*.

RDV graphs were introduced by Gavril [29]; many properties have been discovered and

for many problems efficient algorithms have been found for RDV graphs [1, 42, 43, 58], quite frequently in contrast to only slightly bigger graph classes such as path graphs where the problem turns out to be hard. RDV graphs are the central focus of this thesis, we devote Chapter 3 and Chapter 4 to design sublinear algorithms for two fundamental graph algorithmic problems: maximum matching and minimum dominating set.

RDV graphs can be recognized in linear time, and along the way an RDV representation is produced [29, 21]. Throughout this thesis, we assume that an RDV graph is given in the form of an $O(n)$ sized intersection representation (such a representation always exists, see Lemma 2.3.1 for a proof). Further, we will assume that we are given access to the RDV representation in the form of the host tree T and top and bottom endpoints of paths corresponding to every vertex of the RDV graph.

RDV graphs are strongly chordal. In fact, any bottom-up enumeration of an RDV graph is a strong perfect elimination order, see Lemma 2.3.2 for a proof. A bottom-up enumeration of the vertices in an RDV graph corresponds to sorting the vertices by decreasing depth of $t(v)$. This order can be computed in $O(|T| + n)$ time with bucket-sort. Strongly chordal graphs strictly contain RDV graphs as a subclass, that is, there are graphs which are strongly chordal but not RDV.

Claim 2.2.1. *The graph in Figure 2.3 is strongly chordal but not an RDV graph.*

Proof. Cameron, Hoang, and Leveque [8, Figure 5] showed that the graph in Figure 2.3 is chordal but not a path graph, which in particular implies that it is not an RDV graph.

A result by Farber [24] states that a graph G is strongly chordal if and only if it is chordal and every even-length cycle of length 6 or more in G contains an *odd chord*. Here, an *odd chord* is defined as a chord of a cycle that connects two vertices at an odd distance along the cycle. For example, in Figure 2.3, the edge (v_4, v_7) is an odd chord of the 6-cycle $v_1, v_4, v_5, v_8, v_7, v_6$. A simple exhaustive search of all even-length cycles confirms that the graph in Figure 2.3 is strongly chordal (see code and output in Appendix A). $\square$

Clearly, RDV graphs contain interval graphs as a subclass. To see that this is a strict subclass, consider the RDV graph depicted in Figure 2.4.

Claim 2.2.2. *The graph in Figure 2.4 is an RDV graph but not an interval graph.*

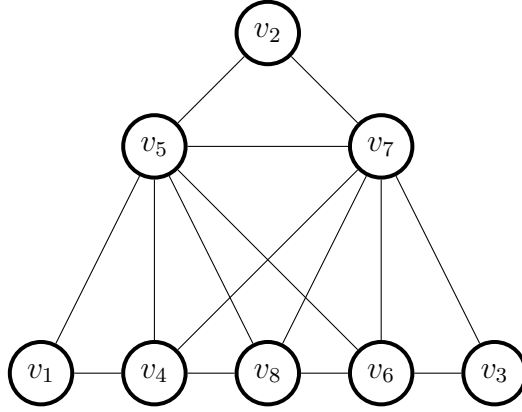


Figure 2.3: An example of a strongly chordal graph that is not an RDV graph.

Proof. The graph has an RDV representation (see Figure 1.2) so it is an RDV graph. The vertices $\{v_2, v_3, v_7\}$ in Figure 2.4 form an *asteroidal triple*, meaning that any two can be connected by a path that avoids the neighborhood of the third. Since interval graphs do not admit asteroidal triples [40], it follows that the graph in Figure 2.4 is not an interval graph. $\square$

It is also worth examining the relationship between RDV graphs and two other well-studied graph classes: *circular arc graphs* and *permutation graphs*. A *circular arc graph* is the intersection graph of arcs on a circle, while a *permutation graph* is a graph whose vertices correspond to elements of a permutation of $[n]$, with edges between pairs of elements that appear in reversed order. Efficient maximum matching algorithms are known for both of these graph classes [41, 60]. However, these results do not directly impact our findings for RDV graphs in Chapter 3, as we will argue that the graph in Figure 2.4 belongs to neither class.

Claim 2.2.3. *The RDV graph in Figure 2.4 is neither a circular arc graph nor a permutation graph.*

Proof. Consider the graph $G' = G \setminus \{v_1\}$. It is known that G' serves as an obstruction for *Helly circular-arc graphs* [44], meaning that it lacks a circular arc representation in which every clique C has its arcs sharing a common point. Since $\{v_4, v_5, v_6\}$ is the only clique of size greater than two, any circular arc representation of G' must place the arcs of

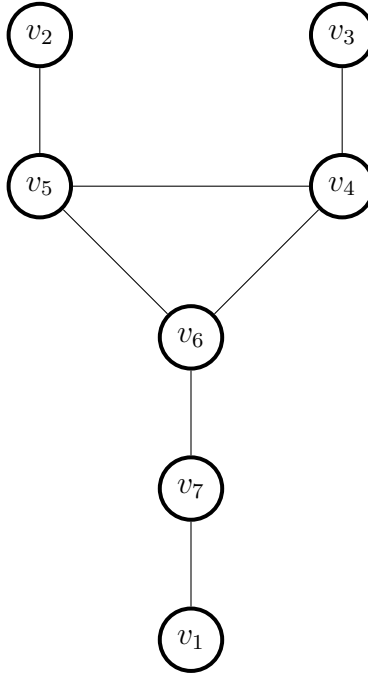


Figure 2.4: An example of an RDV graph that is not an interval graph, circular arc graph or permutation graph. The RDV representation of this graph is given in Figure 1.2.

v_4, v_5 , and v_6 without a shared intersection point. To ensure pairwise intersections, these three arcs must collectively cover the entire circle. However, this configuration prevents the addition of an arc for v_1 , as it has no edges to any of these three vertices. Thus, G is not a circular arc graph.

Furthermore, G' is known not to be a *comparability graph* [33, Figure 5.1], which is a superclass of permutation graphs. Since permutation graphs are closed under vertex deletions, it follows that G cannot be a permutation graph. $\square$

2.2.5 Interval Graphs

Now finally consider graphs with a tree intersection representation where the host tree T is restricted to a path. This graph class consists of intersection graph of subpaths in a path. This is the class of *interval graphs*, which are intersection graphs of intervals on a line. Clearly, interval graphs are RDV graphs, and hence are also strongly chordal and chordal.

See Figure 2.5 for a schematic illustrating the inclusion relationships among some of the graph classes covered in this chapter.

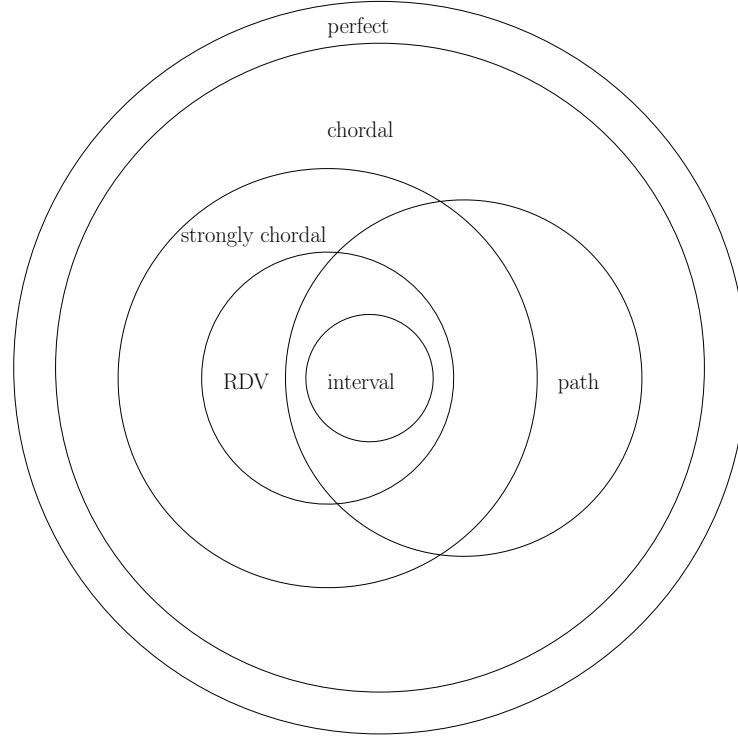


Figure 2.5: Schematic illustrating inclusion relationships among different graph classes

2.3 Tools for RDV graphs

In this section, we briefly prove two things that are easy to show for RDV graphs (but as we have not been able to find citations that state this explicitly, we provide short proofs). First, we may assume that the RDV representation's host tree has at most $O(n)$ nodes. We outline a simple crude upper bound.

Lemma 2.3.1. *Let G be an RDV graph on n vertices. Then G has an RDV representation with host tree T such that $|T| \in O(n)$.*

Proof. As G is an RDV graph, it has an RDV representation with host tree T . If $|T| > 2n + 1$ then some non-root node is neither $t(v)$ nor $b(v)$ for any vertex v of G , and therefore could be combined with its parent without affecting the represented graph. We can repeat this process until $|T| \leq 2n + 1$. $\square$

Second, a bottom-up enumeration of a RDV representation is a strong perfect elimination order. Recall that a bottom-up enumeration is the vertex order obtained by sorting the vertices by decreasing depth of $t(v)$.

Lemma 2.3.2. *Let G be an RDV graph with a given RDV representation. Then any bottom-up enumeration of G is a strong perfect elimination order.*

Proof. Let $v_1, v_2, \dots, v_n$ be a bottom-up enumeration of G . First, observe that since all paths are downward if $v_a \in N[v_b]$ for any two vertices with $a \leq b$, then the top node of $P(v_a)$ is in $P(v_b)$, that is $t(v_a) \in P(v_b)$. Recalling the definition of a strong perfect elimination order, let $i, j, k, l \in [n]$ be arbitrary numbers with $i \leq j$ and $k \leq l$. Now, if $v_k, v_l \in N[v_i]$ and $v_k \in N[v_j]$, we need to show that $v_l \in N[v_j]$. Without loss of generality (as the strongly chordal condition is symmetric for i and k), assume that $i \leq k$. Using the observation, we get that $t(v_i) \in P(v_k)$, $t(v_i) \in P(v_l)$, which in turn implies $v_k \in N[v_l]$ and $t(v_k) \in P(v_l)$. We finish by considering two cases, depending on the order between v_j and v_k .

Case 1: $k \leq j$. This implies that $t(v_k) \in P(v_j)$. As we already know that $t(v_k) \in P(v_l)$, this gives us $v_l \in N[v_j]$ as required.

Case 2: $j < k$. This implies that $t(v_j) \in P(v_k)$. From earlier, we know that $t(v_k), t(v_i) \in P(v_l)$. Since $i \leq j < k \leq l$, we will first encounter $t(v_k)$ and then $t(v_i)$ when walking along $P(v_l)$ starting from $t(v_l)$ (since the bottom-up enumeration is decreasing distance from root). We also know that $t(v_i) \in P(v_k)$. Let P' denote the subpath of $P(v_k)$ between $t(v_k)$ and $t(v_i)$. Then, $P' \subseteq P(v_l)$. Now consider $t(v_j)$. As $i \leq j$, $t(v_j)$ is closer to (or equal) in distance to the root than $t(v_i)$. But, we also know that $t(v_j) \in P(v_k)$. This implies that $t(v_j) \in P'$, which in turn implies $t(v_j) \in P(v_l)$ and $v_l \in N[v_j]$ as required. $\square$

2.4 Data Structures from Computational Geometry

In this section, we introduce two data structures from computational geometry which will later be used in Chapter 3 and Chapter 4.

2.4.1 Priority Search Trees

Priority search trees were introduced by McCreight [46]. This is a dynamic data structure storing tuples of the form $[x, w]$ (we call their aspects *value* and *weight*, respectively), and supports certain queries on the dataset. In particular, we will use the *range minimum* and *range maximum* query.

Definition 2.4.1 (Range Minimum/Maximum Query). *Given a priority search tree $\mathcal{P}$ storing tuples of the form $[x, w]$, a range minimum/maximum query asks for the tuple within a specified range of values $x' \leq x \leq x''$ that has the smallest or largest weight. That is, it returns the tuple $[x, w] \in \mathcal{P}$ such that $x' \leq x \leq x''$ and w is minimized or maximized.*

Priority search trees support insertion and deletion of a tuple as well as range minimum/maximum queries in $O(\log n)$ time per query, where n is the number of tuples stored in the priority search tree. We will use priority search trees in Chapter 3 and Chapter 4.

2.4.2 Dynamic Orthogonal Ray Shooting

The other data structure supports storing segments and querying with an orthogonal ray.

Definition 2.4.2 (Orthogonal Ray Shooting Query). *Given a set S of horizontal line segments in the plane, an orthogonal ray shooting query determines the first horizontal segment hit by a vertical query ray $\vec{q}$. The vertical ray $\vec{q}$ originates from a specified point and extends infinitely upward.*

The *orthogonal ray shooting problem* is as follows: design a dynamic data structure to maintain a set S of horizontal line segments in order to support orthogonal ray shooting queries. Many data structures have been developed for this problem (some for more general versions), see for example [48], [39] for older results with slower processing time.

The current best run-time bounds are by Giyora and Kaplan [32], who showed how to implement all three operations (insertions, deletions and ray shooting queries) in $O(\log n)$ time, using $O(n)$ space, where n is the number of horizontal segments stored. Later on this was generalized to drop the requirement of orthogonality [55] without affecting space or runtime. For our purposes in Chapter 4, we will use the data structure of Giyora and Kaplan.

Chapter 3

Maximum Matching

3.1 Introduction

Maximum matching is one of the oldest problems in the history of graph theory and graph algorithms: Given a graph $G = (V, E)$, find a *matching* (a set of pairwise non-adjacent edges) that is *maximum* (has the largest possible number of edges). See for example extensive reviews of the older history of matchings and its applications in [3, 45]. The fastest known algorithm for general graphs runs in $O(\sqrt{nm})$ time ([50], see also [64]). There have been some recent breakthroughs for algorithms for maximum flow, culminating in an algorithm with almost-linear run-time $O(m^{1+o(1)})$ [13]; this immediately implies an almost-linear algorithm for maximum matching in bipartite graphs.

What about truly linear-time algorithms, i.e., those with $O(m + n)$ run-time, at least for graphs with certain special properties? A natural approach is to use a greedy matching algorithm (see Section 3.2 for details). While in general the resulting matching need not be optimal, for certain graph classes running the greedy algorithm on a suitably chosen vertex order returns the optimal maximum matching.

One graph class where this can be done is the interval graphs. Moitra and Johnson [51] proved that the greedy algorithm always computes a maximum matching in an interval graph when executed on the vertex order that is obtained by sorting the intervals by their right endpoints. Given an interval representation where all endpoints have integer coordinates in $O(n)$, this gives an $O(m + n)$ algorithm for interval graphs. Liang and Rhee

[41] improve this further to $O(n \log n)$ time by using binary search trees; in particular this is *sub-linear* run-time if the graph has $\omega(n \log n)$ edges. This runtime can be easily improved to $O(n \log \log n)$ by using a van Emde Boas tree [63], as observed later by Liang and Rhee [60].

This greedy strategy actually works for strongly chordal graphs too when executed on a strong perfect elimination order, as shown by Dahlhaus and Karpinski [19]. A straightforward implementation runs in $O(m + n)$ time. Since RDV graphs are strongly chordal, can we use the intersection representation of RDV graphs to achieve an $O(n \log n)$ runtime, similar to interval graphs? We answer this positively in the rest of this chapter.

Our idea is to use the greedy-algorithm, and to speed up the time it takes to find a minimum (in the vertex ordering) unmatched neighbor to $O(\log n)$ per query. The key ingredient here is that “is v_i a neighbor of v_j ” (for some $i < j$) can be re-phrased, using the RDV representation, as the question of checking whether a horizontal segment (corresponding to v_i) intersects a vertical segment (corresponding to v_j). See Section 3.4 and Section 3.5 for more details. This turns the question of finding v_i ’s minimum unmatched neighbor (in the vertex order) into a range minimum query. By using the data structures from Section 2.4 suitably, we can answer these queries efficiently.

There are a number of other results concerning fast algorithms to solve maximum matching in intersection graphs of some geometric objects (see Table 3.1). Specifically, the results for interval graphs were extended to *circular arc graphs* (intersection graphs of arcs of a circle) [41]. In an entirely different approach, maximum matching can also be solved very efficiently in *permutation graphs* [60]; see also [22] for an (unpublished) matching-algorithm for permutation graphs that is slower but beautifully uses range queries to find the matching. Permutation graphs are a special case of *co-comparability graphs*, i.e., graphs for which the complement has an acyclic transitive orientation; these can also be viewed as intersections of curves between two parallel lines [34]. For these, maximum matchings can be found in linear time [49].

As noted earlier in Chapter 2, RDV graphs are unrelated to circular arc graphs or permutation graphs, so these results do not directly impact ours or vice versa.

The question whether chordal graphs have a linear-time algorithm for maximum matching remains open. In fact, no algorithms faster than those for general graphs are known for chordal graphs (or even path graphs). But likely the answer is “no”, because as argued in [19], a linear-time algorithm for testing the existence of a *perfect matching* (i.e., a matching

of size $n/2$) in a chordal graph would imply a linear-time algorithm for the same problem in any bipartite graph that is *dense* (has $\Theta(n^2)$ edges).

Graph Class	Runtime	Reference
Interval graphs	$O(n \log \log n)$	[41, 60]
Circular arc graphs	$O(n \log n)$	[41]
Permutation graphs	$O(n \log \log n)$	[60]
Strongly chordal graphs	$O(n + m)$	[19]
Co-comparability graphs	$O(n + m)$	[49]
RDV graphs	$O(n \log n)$	Section 3.5

Table 3.1: Existing and new results for maximum matching in some classes of graphs, presuming a suitable intersection representation is given and sufficiently small.

This chapter is structured as follows. We begin by presenting a greedy algorithm for matching and providing an alternative proof of its optimality for strongly chordal graphs. We then introduce additional notation for RDV graphs, defining horizontal and vertical segments for each vertex. Next, we establish our main theorem, which reduces adjacency queries to detecting intersections between these segments. Finally, we leverage this theorem to speed up the greedy algorithm, achieving an $O(n \log n)$ runtime.

3.2 Greedy Strategy for matching

In its most general setting, a greedy strategy simply adds edges to a matching M until no edge has two *unmatched* endpoints, i.e., endpoints not incident to an edge in M . The specific greedy strategy we consider here is a bit more sophisticated, parsing vertices in a specific order and breaking ties among possible edges to add by the indices in the order. So consider a graph G with some vertex order $v_1, v_2, \dots, v_n$. The main idea behind Algorithm 1 is as follows. The algorithm begins by initializing the matching M as empty and processes the vertices in the vertex order. For each vertex v_i being processed,

1. If v_i has been matched earlier, do nothing.
2. Otherwise, if v_i has unmatched neighbors, select the unmatched vertex $v_j \in N(v_i)$ that minimizes its index j in the vertex order, and add the edge (v_i, v_j) to the matching: $M \leftarrow M \cup \{(v_i, v_j)\}$

The pseudocode for this algorithm is presented in Algorithm 1. Note that this algorithm can be run on any vertex order and it will always return a valid maximal matching, although it may not necessarily be a maximum matching.

Algorithm 1: Greedy-algorithm for matching

Input: A graph G with a vertex order $v_1, \dots, v_n$

```

1 Initialize the matching  $M \leftarrow \emptyset$ 
2 Initialize all vertices to be “unmatched”
3 for  $i = 1, \dots, n$  do
4     if  $v_i$  is “unmatched” and has “unmatched” neighbors then
5         Find the “unmatched” vertex  $v_j \in N(v_i)$  that minimizes  $j$ .
           //  $j > i$ , for otherwise  $v_j$  would have been matched earlier
6          $M \leftarrow M \cup \{(v_i, v_j)\}$ 
7         Mark  $v_i$  and  $v_j$  as “matched”
8 return  $M$ 

```

3.3 Optimality for strongly chordal graphs

Dahlhaus and Karpinski [19] showed that Algorithm 1, when run on a strong perfect elimination order of a strongly chordal graph, returns a maximum matching by way of a potential function argument. Later, Campelo and Klein [9] re-proved this result (while considering vertex-weighted matching in strongly chordal graphs) using a linear programming formulation. In this section, we provide an alternate inductive proof.

Theorem 3.3.1. [19] *Algorithm 1 returns a maximum matching when run on a strong perfect elimination order of a strongly chordal graph.*

Proof. We first set up some notation. Let M_i be the (partial) matching constructed by the algorithm after processing vertex v_i (the algorithm will return M_n), and define M_0 to be the empty matching. We will also call two matchings A and B as behaving *identically* on $\{v_1, \dots, v_i\}$ if for any vertex v_x with $x \in [i]$, either v_x is unmatched in both A and B , or it is matched to the same vertex in both A and B . We use the notation $A \sim_i B$ to denote that matchings A and B behave identically on $\{v_1, \dots, v_i\}$.

The main ingredient of the proof is to show a suitable invariant: M_i behaves identically for vertices $\{v_1, \dots, v_i\}$ as some maximum matching M_i^* .

Claim 3.3.2. *Let M_i be the partial matching after processing v_i . Then, there exists a maximum matching M_i^* such that both M_i and M_i^* behave identically on $\{v_1, \dots, v_i\}$.*

Proof. An *alternating path* with respect to a matching M is a (possibly empty) path that starts at an unmatched vertex and alternates between unmatched and matched edges. Clearly, if P is an alternating path of even length, then switching the unmatched and matched edges in P produces a new matching of the same size. Similarly, switching an *alternating cycle* of even length results in a matching of equal size.

We prove the claim using induction on i . The base case $i = 0$ is trivial, since M_0 is empty. Let us assume that Claim 3.3.2 holds for $0, 1, \dots, i - 1$ for some $i \geq 1$; we will show that it holds for i as well.

By induction hypothesis, we know that $M_{i-1} \sim_{i-1} M_{i-1}^*$. At the current iteration, we are processing v_i , and we need to find a maximum matching M_i^* such that $M_i \sim_i M_i^*$. We proceed by breaking into cases.

Case 1: v_i is matched by M_{i-1} . In this case, the partial matching will be unchanged, that is, $M_i = M_{i-1}$. Thus, we define $M_i^* = M_{i-1}^*$. As v_i was already matched by M_{i-1} to a vertex in $v_1, \dots, v_{i-1}$, this ensures that $M_i \sim_i M_i^*$.

Case 2: v_i is not matched by M_{i-1} . In this case, the algorithm will select the unmatched vertex $v_j \in N(v_i)$ (if such a vertex exists) that minimizes j (in particular, $j > i$), and add (v_i, v_j) to the partial matching. If the algorithm is unable to match v_i , this means that v_i was unmatched in M_{i-1}^* too. In that case, we are done since $M_i = M_{i-1}$ and we can define $M_i^* = M_{i-1}^*$. So from now on, we will assume that the algorithm matches v_i with v_j . That is, $M_i = M_{i-1} \cup \{(v_i, v_j)\}$. We proceed by analyzing sub-cases based on whether v_i is matched in M_{i-1}^* .

Case 2.1: v_i is unmatched by M_{i-1}^* . Since M_{i-1}^* is a maximum matching, v_j must be matched by M_{i-1}^* (otherwise, we could add (v_i, v_j) to M_{i-1}^* , thereby increasing its size). Let v_j be matched to v_k by M_{i-1}^* . Clearly, $k \geq i$ because $M_{i-1}^* \sim_{i-1} M_{i-1}$ and v_j was unmatched by M_{i-1} . Furthermore, $k \neq i$ since v_i is unmatched by M_{i-1}^* . We define $M_i^* = M_{i-1}^* \cup \{(v_i, v_j)\} \setminus \{(v_j, v_k)\}$. Clearly, M_i^* is a maximum matching, as we obtain it by switching the alternating path v_i, v_j, v_k in M_{i-1}^* . It is straightforward to verify that we have $M_i \sim_i M_i^*$.

Case 2.2: v_i is matched by M_{i-1}^* . Let v_i be matched to v_l by M_{i-1}^* . Clearly, $l > i$ as $M_{i-1} \sim_{i-1} M_{i-1}^*$ and v_i was unmatched by M_{i-1} . Since the algorithm chose v_j instead of v_l , we must have $j \leq l$. If $j = l$ then we can define $M_i^* = M_{i-1}^*$. So from this point on, we will assume $j \neq l$. We consider two sub-cases based on whether v_j (which the algorithm chose) is matched by M_{i-1}^* or not.

Case 2.2.1: v_j is not matched by M_{i-1}^* . We define $M_i^* = M_{i-1}^* \cup \{(v_i, v_j)\} \setminus \{(v_i, v_l)\}$. Clearly, M_i^* is a maximum matching since we obtain it by switching the alternating path v_j, v_i, v_l in M_{i-1}^* . It is easy to verify that $M_i \sim_i M_i^*$.

Case 2.2.2: v_j is matched by M_{i-1}^* . Let v_j be matched to v_k by M_{i-1}^* . Clearly, $k \neq l$ since v_k and v_l are matched to different vertices. Further, $i \leq k$ as $M_{i-1} \sim_{i-1} M_{i-1}^*$ and v_j was unmatched by M_{i-1} . Thus, we know that $i \leq k$, $j \leq l$, $v_i, v_l \in N[v_j]$ and $v_j \in N[v_k]$. Using the strongly chordal property (by relabeling j to k and vice versa), this implies that $v_k \in N[v_l]$. Hence, (v_k, v_l) is an edge. Thus, i, j, k, l are all distinct and v_i, v_j, v_k, v_l is an alternating cycle for M_{i-1}^* . We define $M_i^* = M_{i-1}^* \cup \{(v_i, v_j), (v_k, v_l)\} \setminus \{(v_i, v_l), (v_j, v_k)\}$. It is easy to verify that $M_i \sim_i M_i^*$. $\square$

To finish, observe that as a direct consequence of Claim 3.3.2 there is some maximum matching M_n^* which “behaves identically” for all vertices as M_n . This implies that M_n (which the algorithm returns) is a maximum matching. $\square$

3.4 Neighborhood queries in RDV graphs

Our key idea for a fast implementation of greedy algorithms is that adjacency queries in an RDV graph can be reduced to the question of whether a horizontal segment intersects a vertical segment. We need some additional definitions. First, it will be convenient to assign points to the nodes of the host tree T as follows¹.

Definition 3.4.1 (Drawing the host tree T). *To draw a tree T , fix an arbitrary order of children at each node, and then enumerate the leaves of T as $L_1, \dots, L_\ell$ from left to right. For every node i in T , let $\ell(i)$ be the leftmost (i.e., lowest-indexed) leaf that is a descendant of i , and set $x(i)$ to be the index of $\ell(i)$. We also need the notation $r(i)$ for the rightmost*

¹This definition isn’t meant to create a visually appealing drawing but rather to assign coordinates to the tree’s nodes for later use.

leaf that is a descendant of i . Also, define $y(i)$ to be the distance of node i from the root of T . Then, we draw the node $i \in T$ at point $(x(i), y(i))$.

Figure 3.1 shows an example of such a drawing (where y -coordinates increase top-to-bottom). We can compute $x(\cdot)$ with a post-order traversal and $y(\cdot)$ with a BFS traversal of host-tree T in $O(|T|)$ time.

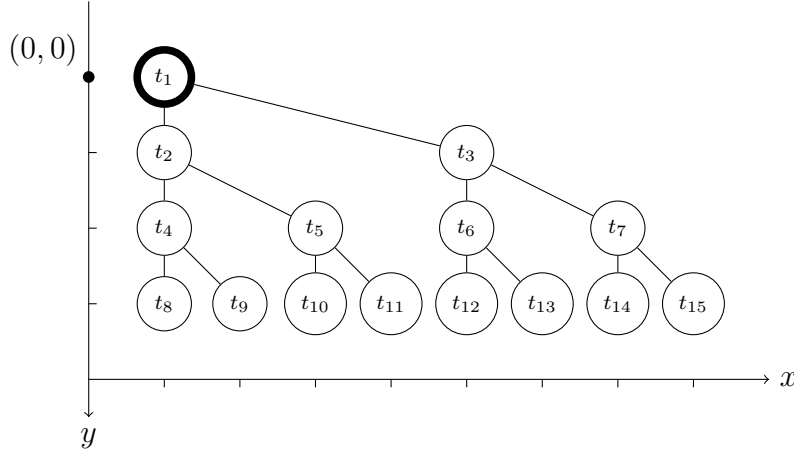


Figure 3.1: Drawing the host tree T according to Definition 3.4.1.

Second, we define two segments for each vertex of the RDV graph: one horizontal and one vertical. Recall that $t(v)$ and $b(v)$ were the top and bottom nodes of the path $P(v)$ corresponding to vertex v .

Definition 3.4.2. Let G be a graph with an RDV representation, with the nodes of the host tree assigned points according to Definition 3.4.1. For each vertex v , define the following (see Figure 3.2 for examples):

- The horizontal segment $\mathbf{h}(v)$ of v is the segment between the point of $t(v)$ and $(x(r(t(v))), y(t(v)))$, i.e., it extends rightward from the point of $t(v)$ until it is above the rightmost descendant of $t(v)$.
- The vertical segment $\mathbf{v}(v)$ of v is the segment between the point of $b(v)$ and $(x(b(v)), y(t(v)))$, i.e., it extends upward from the point of $b(v)$ until it is to the right of $t(v)$.

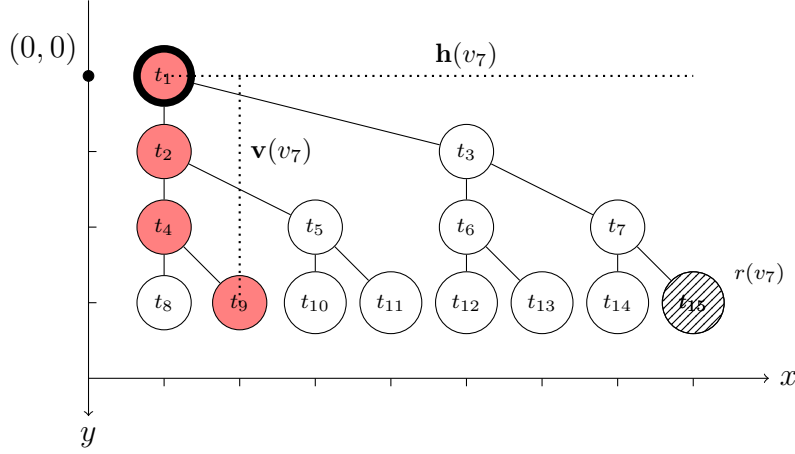


Figure 3.2: The host tree T from Figure 1.2, with the path P_7 (corresponding to vertex v_7) highlighted in red and the horizontal segment $\mathbf{h}(v_7)$ and vertical segment $\mathbf{v}(v_7)$ are shown. $r(v_7)$, the rightmost descendant of $t(v_7)$, is highlighted with hatched lines.

Refer to Figure 3.2 for an example illustrating the two previously mentioned segments.

We now show that adjacency queries in an RDV graph correspond to checking whether orthogonal segments intersect.

Lemma 3.4.1. *Let G be a graph with an RDV representation and let $v_1, \dots, v_n$ be a bottom-up enumeration of vertices. Further, for any vertex v , let $\mathbf{h}(v)$ and $\mathbf{v}(v)$ be the horizontal and vertical segments from Definition 3.4.2. Then for any $i < j$, edge (v_i, v_j) exists if and only if the vertical segment $\mathbf{v}(v_j)$ intersects the horizontal segment $\mathbf{h}(v_i)$.*

Proof. Since $\mathbf{v}(v_j)$ is a vertical segment and $\mathbf{h}(v_i)$ is a horizontal segment, they intersect if and only if both the x -coordinates and y -coordinates line up correctly, i.e., $x(t(v_i)) = x(\ell(t(v_i))) \leq x(b(v_j)) \leq x(r(t(v_i)))$ and $y(t(v_j)) \leq y(t(v_i)) \leq y(b(v_j))$.

Assume first that edge (v_i, v_j) exists, which means that $P(v_i)$ and $P(v_j)$ have a node u in common. Among all such nodes u , pick the one that minimizes the y -coordinate; this implies $u \in \{t(v_i), t(v_j)\}$. By $i < j$ we actually know that $u = t(v_i)$, because if $u \neq t(v_i)$ then $u = t(v_j)$ would be a strict descendant of $t(v_i)$ and have larger y -coordinate, contradicting the bottom-up enumeration ordering. Since $u \in P(v_j)$, node $b(v_j)$ is a descendant of u (which in turn is a descendant of $t(v_j)$), so $y(t(v_j)) \leq y(u) = y(t(v_i)) \leq$

$y(b(v_j))$ and the y -coordinates line up. The x -coordinates line up since $b(v_j)$ is a descendant of $t(v_i)=u$ and the horizontal segment $\mathbf{h}(v_i)$ covers all such descendants.

Assume now that the segments intersect. By $y(t(v_j)) \leq y(t(v_i)) \leq y(b(v_j))$ the path $P(v_j)$ contains a node (call it u) with $y(u) = y(t(v_i))$. If u equals $t(v_i)$ then $P(v_i)$ and $P(v_j)$ have node $t(v_i)$ in common and (v_i, v_j) is an edge as desired. If $u \neq t(v_i)$, then these two nodes (with the same y -coordinate) have a disjoint set of descendants, so the intervals $I_u = [x(\ell(u)), x(r(u))]$ and $I_i = [x(\ell(t(v_i))), x(r(t(v_i)))]$ are disjoint. Since $b(v_j)$ is a descendant of $u \in P_j$, we have $x(b(v_j)) \in I_u$, but since the x -coordinates line up we have $x(b(v_j)) \in I_i$. This is impossible. $\square$

Theorem 3.4.2. *Let G be a graph with an RDV representation. Then, for any two vertices $v \neq w$, (v, w) is an edge if and only if $\mathbf{h}(v) \cap \mathbf{v}(w) \neq \emptyset$ or $\mathbf{h}(w) \cap \mathbf{v}(v) \neq \emptyset$ or both.*

Alternately, $v \in N[w]$ if and only if $\mathbf{h}(v) \cap \mathbf{v}(w) \neq \emptyset$ or $\mathbf{h}(w) \cap \mathbf{v}(v) \neq \emptyset$ or both.

Proof. Fix a bottom-up enumeration $v_1, v_2, \dots, v_n$. Up to renaming, we have $v = v_i$ and $w = v_j$ for some $i < j$. For the forward direction, if (v, w) is an edge then by Lemma 3.4.1 $\mathbf{h}(v) \cap \mathbf{v}(w) \neq \emptyset$. For the backward direction, there are two cases. **Case 1:** $\mathbf{h}(v) \cap \mathbf{v}(w) \neq \emptyset$. Then applying Lemma 3.4.1, we get that (v, w) is an edge. **Case 2:** $\mathbf{h}(w) \cap \mathbf{v}(v) \neq \emptyset$. As w appears after v in the bottom up enumeration order, the only way $\mathbf{h}(w)$ and $\mathbf{v}(v)$ can intersect is if $t(w)$ and $t(v)$ are at equal distance from the root of the host tree T . Since we break ties arbitrarily while bottom-up enumerating (as in Definition 2.2.3), we can swap the positions of w and v in the bottom-up enumeration. Then, Lemma 3.4.1 implies that (v, w) is an edge.

The alternate statement of the theorem follows directly since for every vertex v , we always have $\mathbf{h}(v) \cap \mathbf{v}(v) \neq \emptyset$. $\square$

Theorem 3.4.2 provides an *implicit representation* of the RDV graph class. Specifically, for an n -vertex graph $G \in \mathcal{F}$ from some family $\mathcal{F}$, an implicit representation consists of an assignment of $O(\log n)$ -bit codes to the vertices of G along with a decoder function f that, given the codes of two vertices, determines adjacency, where f is defined independently of the specific choice of $G \in \mathcal{F}$. Theorem 3.4.2 shows for an RDV graph we can encode every vertex with 5 $O(n)$ numbers (corresponding to the 5 coordinates required to specify the horizontal and vertical segments). While an implicit representation for path graphs was previously known [38], it is insufficient for our purposes, as it does not allow efficient

neighborhood searches for a given vertex—unlike the representation provided by Theorem 3.4.2.

Theorem 3.4.2 also shows that RDV graphs are *T-intersection graphs*, which are intersection graphs of *T* shaped objects in the plane. These graphs have been previously studied in the literature. For instance, it is known that planar graphs are *T*-intersection graphs [20]. In fact, this paper shows that planar graphs can be represented by *T* shaped objects that can only touch, i.e., they are interior disjoint. If we allow the objects to intersect in the interior, then even *L* shaped objects are enough to represent planar graphs [35].

3.5 Sublinear $O(n \log n)$ matching algorithm

In light of Lemma 3.4.1 and Theorem 3.4.2, we can reformulate the greedy algorithm and speed it up by using range minimum queries of priority search trees, which were defined earlier in Section 2.4.1.

Definition 3.5.1 (tuple of a vertex). *For a vertex v_i , we refer to the value-weight tuple $[x(b(v_i)), i]$ as the tuple of v_i .*

We store tuples in a priority search tree $\mathcal{P}$ according to the following invariant.

Invariant 1: While processing vertex v_i , the priority search tree $\mathcal{P}$ should only store tuples of unmatched vertices v_t (for $t > i$) for which $y(b(v_t)) \geq y(t(v_i)) \geq y(t(v_t))$, i.e., all unmatched vertices ahead of v_i in the vertex order whose vertical segments encompass the current y -coordinate within their y -range.

Lemma 3.5.1 (Efficient neighborhood queries in RDV graphs). *Assuming a suitable priority search tree is maintained according to Invariant 1, in Algorithm 1, one can find unmatched vertex $v_j \in N(v_i)$ that minimizes the index j in $O(\log n)$ time.*

Proof. By Lemma 3.4.1, to find unmatched vertex $v_j \in N(v_i)$ which minimizes j , we can simply perform a range minimum query with the x -range of $\mathbf{h}(v_i)$. Invariant 1 ensures that the y -coordinates of $\mathbf{h}(v_i)$ and $\mathbf{v}(v_j)$ line up, and the query range ensures that the x -coordinates do, which means that any tuple with value in the query range belongs to a vertex that is (i) unmatched and (ii) in $N(v_i)$. Among such vertices, the one with minimum weight j is the smallest one in the elimination order (as is required by Algorithm 1). Since we need to perform one range minimum query, this takes $O(\log n)$ time. $\square$

Algorithm 2 shows how to implement this idea. For convenience, assume that the range minimum query returns a null value if no tuple exists within the specified range, and define a sentinel value $y(t(v_0)) = +\inf$. Also, let $\mathbf{h}(v_i)|_x = [x(t(v_i)), x(r(t(v_i)))]$, i.e., the x -coordinate range of the horizontal segment $\mathbf{h}(v_i)$ be the *projection* of $\mathbf{h}(v_i)$ on the x axis.

Algorithm 2: Reformulated Greedy-algorithm for matching in RDV graphs

Input: An RDV graph G with a bottom-up enumeration order $v_1, \dots, v_n$

```

1 Initialize the matching  $M = \emptyset$ 
2 Initialize all vertices to be “unmatched”
3 Initialize an empty priority search tree  $\mathcal{P}$ 
4 for  $i = 1, \dots, n$  do
5   for all “unmatched” vertices  $v_t$  such that  $y(t(v_{i-1})) > y(b(v_t)) \geq y(t(v_i))$  do
6      $\mathcal{P}.\text{insert}([x(b(v_t)), t])$ 
7   if  $v_i$  is “unmatched” then
8      $\mathcal{P}.\text{delete}([x(b(v_i)), i])$ 
9      $[\cdot, j] \leftarrow \mathcal{P}.\text{rangemin}(\mathbf{h}(v_i)|_x)$ 
10    if the result of the range minimum query is not null then
11       $M \leftarrow M \cup \{(v_i, v_j)\}$ 
12      Mark  $v_i$  and  $v_j$  as “matched”
13       $\mathcal{P}.\text{delete}([x(b(v_j)), j])$ 
13 return  $M$ 

```

Lemma 3.5.2. *Algorithm 2 returns a maximum matching and finishes in $O(n \log n)$ time.*

Proof. Let v_t be an arbitrary vertex. Note that we insert the tuple of the unmatched vertex v_t into $\mathcal{P}$ (in lines 4-6) as soon as $y(t(v_i))$ (where v_i is the current vertex) becomes less than or equal to $y(b(v_i))$. Conversely, we remove v_t from $\mathcal{P}$ either when it gets matched or when v_t itself is processed, hence $y(t(v_t))$ is less than or equal to $y(t(v_i))$. As a result, Invariant 1 holds. Due to Invariant 1 and Lemma 3.5.1, we exactly implement the greedy algorithm as in Algorithm 1. By Theorem 3.3.1, Algorithm 2 returns a maximum matching.

For the runtime, observe that we perform one range minimum query per iteration, and each query takes $O(\log n)$ time. Since there are at most n iterations, the queries take

$O(n \log n)$ time. Maintaining an array of $y(b(v_t))$ values for all vertices that has been pre-sorted with bucket sort and marking vertices upon matching allows us to add tuples of eligible unmatched v_t in lines 4-6 in total $O(n)$ time. We may add more than one vertex to $\mathcal{P}$ in some iterations, but overall, every vertex gets added at most once. Similarly, every vertex is removed at most once, so that takes $O(n \log n)$ total time as well. $\square$

With this, we can put everything together into our first major theorem.

Theorem 3.5.3. *Given an n -vertex graph G with an $O(n)$ sized RDV representation T , a maximum matching of G can be found in $O(n \log n)$ time.*

Proof. Parse T to compute the x -coordinates and y -coordinates of all nodes in T , then bucket-sort the vertices by decreasing $y(t(v))$ to obtain the bottom-up enumeration order $v_1, \dots, v_n$ in $O(n)$ time. By Theorem 3.3.1 applying the greedy-algorithm with this vertex-ordering will give a maximum matching. Using Algorithm 2 with a priority search tree, the run-time of the algorithm is reduced to $O(n \log n)$ time in total. $\square$

We end with an open question. Recall that for interval graphs, an improvement of the run-time for matching from $O(n \log n)$ to $O(n \log \log n)$ is possible by using van Emde Boas trees [60]. This naturally leads one to ask: Could the run-time of Theorem 3.3.1 also be improved to $O(n \log \log n)$ time? The bottleneck for this would be to improve the run-time for range minimum queries if all coordinates are (small) integers, e.g. via van Emde Boas trees. To our knowledge, this problem has not been studied.

Chapter 4

Minimum Dominating Set

4.1 Introduction

Given a graph G , a subset S of vertices is called *dominating* if every vertex in the graph is either included in S or adjacent to a vertex in S . A *minimum dominating set* is a dominating set with the fewest vertices. Dominating sets are useful in routing problems in wireless networks and facility location problems. Finding the minimum dominating set is NP-hard in general [27], as there exists a straightforward reduction from the set cover problem. On the other hand, a simple greedy algorithm providing a $\log n$ -approximation is well known [15].

What about RDV graphs and related graph classes? Just as for maximum matching, the restricted structure of these graph classes allows us to design polynomial time (and in some cases linear time) algorithms for the minimum dominating set problem. Polynomial time algorithms exist for permutation graphs [26] and cographs [17]. For strongly chordal graphs, the situation is even better, the minimum dominating set can be found in linear time [25]. The minimum dominating set problem remains NP-Hard for chordal graphs [6]. For more information about the complexity of minimum dominating set and other related problems in different subclasses of perfect graphs, the interested reader can look at a review by Corneil and Stewart [18].

As for minimum dominating set in RDV graphs, an $O(n+m)$ greedy algorithm (reviewed in Algorithm 3) was shown to find the minimum dominating set by Booth and Johnson in

[6]. Later, Farber generalized this result with the same runtime to strongly chordal graphs, and even beyond to restricted weighted versions [25]. For interval graphs, an $O(n)$ greedy algorithm is alluded to in [65] but we have not been able to find the original reference for this result.

Cheng, Kaminski, and Zaks [65] studied minimum dominating sets in intersection graphs of downward paths in a spider graph (which is a tree consisting of multiple paths sharing a common node). This class of graphs sits between interval graphs and RDV graphs. Their approach achieve an $O(n)$ runtime for this graph class, by using the $O(n)$ algorithm for interval graphs.

In the following sections, we will first present the greedy strategy to create a dominating set. Then, we will show that this strategy is optimal, meaning that it returns a *minimum* dominating set, for strongly chordal graphs. Booth and Johnson [6] were the first to show that this strategy is optimal, but only for RDV graphs. On the other hand, Farber’s [25] algorithm for dominating sets in strongly chordal graphs employs a different linear programming approach. Our proof generalizes Booth and Johnson’s result to strongly chordal graphs, while closely following their proof strategy. Next, we will present our main result in this chapter — how our techniques can be applied to improve the runtime for RDV graphs to $O(n \log n)$.

4.2 Greedy strategy

Consider a graph G with some vertex order $v_1, v_2, \dots, v_n$. The main idea for finding a dominating set is a simple greedy strategy. The algorithm begins by initializing the dominating set D as empty and processes the vertices in the vertex order. For each vertex v_i being processed:

1. If v_i is dominated by D , do nothing.
2. Otherwise, if v_i is undominated, select the vertex $v_t \in N[v_i]$ that maximizes t (its index in the vertex order), and add v_t to the dominating set: $D \leftarrow D \cup \{v_t\}$.

The pseudocode for this algorithm is presented in Algorithm 3. Note that this algorithm can be run on any vertex order and it will always return a valid dominating set, although

it may not necessarily be a minimum dominating set. Observe that Algorithm 3 for the dominating set problem shares several similarities with Algorithm 1 for matching—both follow a greedy approach based on a specific vertex ordering. However, some differences exist: one selects the maximum vertex in the vertex order while the other chooses the minimum, and they differ in whether marked or dominated vertices are included in the search. Furthermore, in the case of the dominating set, marking neighbors as dominated takes non-trivial time since there could be more than a constant number of them.

Algorithm 3: Greedy strategy for dominating set

Input: A graph G with a vertex order $v_1, \dots, v_n$

```

1 Initialize the dominating set  $D = \{\}$ 
2 Mark all vertices as “undominated”.
3 for  $i = 1, 2, \dots, n$  do
4   if  $v_i$  is “undominated” then
5     Find the vertex  $v_t \in N[v_i]$  that maximizes  $t$ .
6      $D \leftarrow D \cup \{v_t\}$ 
7     Mark all vertices in  $N[v_t]$  as being “dominated”.
8 return  $D$ 

```

4.3 Optimality for strongly chordal graphs

The greedy strategy presented in Algorithm 3 returns a *minimum* dominating set when run on a strong perfect elimination order of a strongly chordal graph. In this section, we present a proof of this, following the proof strategy of Booth and Johnson, whose original result was only for RDV graphs, but as we will see, generalizes naturally.

Theorem 4.3.1. *If applied to a strongly perfect elimination order of a strongly chordal graph, Algorithm 3 returns a minimum dominating set of G .*

Proof. Let us set up some notation. Let D_i be the (partial) dominating set constructed by the algorithm after processing vertex v_i (the algorithm will return D_n). Also define $D_0 = \{\}$. Further, let $\text{DOM}(S)$ denote the vertices dominated by some set $S \subseteq V(G)$.

Our first claim follows directly from the algorithm.

Claim 4.3.2. *Let D_i be the partial dominating set after processing v_i . Then, vertices $v_1, v_2, \dots, v_i$ are dominated by D_i . That is, for all $i \in \{0, 1, 2, \dots, n\}$, we have $\{v_1, v_2, \dots, v_i\} \subseteq \text{DOM}(D_i)$.*

The main ingredient of the proof is to show a suitable invariant: D_i is a subset of some minimum dominating set of G .

Claim 4.3.3 (invariant lemma). *Let D_i be the partial dominating set after processing v_i . Then, for all $i \in \{0, 1, 2, \dots, n\}$, there exists a minimum dominating set of G , say D_i^* , such that $D_i \subseteq D_i^*$.*

Proof. We proceed using induction on i . The base case $i = 0$ is trivial, since D_0 is empty. Let us assume that Claim 4.3.3 holds for $0, 1, \dots, i-1$ for some $i \geq 1$; we will show that it holds for i as well.

By induction hypothesis, we know that $D_{i-1} \subseteq D_{i-1}^*$ at the previous iteration. At the current iteration, we are processing v_i , and we need to find a minimum dominating set D_i^* such that $D_i \subseteq D_i^*$. We proceed by breaking into cases.

Case 1: $v_i \in \text{DOM}(D_{i-1})$. In this case, v_i is already dominated, so the partial dominating set will be unchanged, that is, $D_i = D_{i-1}$. Thus, we define $D_i^* = D_{i-1}^*$ which ensures that $D_i \subseteq D_i^*$.

Case 2: $v_i \notin \text{DOM}(D_{i-1})$. In this case, the algorithm will select $v_t \in N[v_i]$, with highest index t in the elimination order, and add it to the partial dominating set: $D_i = D_{i-1} \cup \{v_t\}$. Further, we know that D_{i-1}^* is a minimum dominating set, thus there is some $v_a \in D_{i-1}^*$ which dominates v_i , that is, $v_i \in N[v_a]$.

Case 2.1: $v_t \in D_{i-1}^*$. In this case, we define $D_i^* = D_{i-1}^*$. Since $D_{i-1} \subseteq D_{i-1}^*$, we know that $D_{i-1} \cup \{v_t\} \subseteq D_{i-1}^* = D_i^*$ and we are done.

Case 2.2: $v_t \notin D_{i-1}^*$, and in particular therefore $v_t \neq v_a$. In this case, we define $D_i^* = D_{i-1}^* \cup \{v_t\} \setminus \{v_a\}$. To see that this satisfies the conditions, first observe that $v_a \notin D_{i-1}$ since v_i is not dominated by D_{i-1} , while from the inductive hypothesis we know that $D_{i-1} \subseteq D_{i-1}^*$ which implies that $D_i \subseteq D_i^*$ since $v_a \notin D_{i-1}$. Secondly, $|D_{i-1}^*| = |D_i^*|$, thus D_i^* has the same size as the minimum dominating set D_{i-1}^* . So to prove that D_i^* is a minimum dominating set all that remains to show is that it dominates every vertex.

Since $D_i \subseteq D_i^*$, it is clear that D_i^* dominates $v_1, v_2, \dots, v_i$, so we only need to prove that it also dominates v_j for $j \in \{i+1, \dots, n\}$ which by $D_i^* = D_{i-1}^* \cup \{v_t\} \setminus \{v_a\}$ is nontrivial only

if $v_j \in N[v_a]$. For such a v_j , we will show that it gets dominated by v_t , that is $v_j \in N[v_t]$. We consider two sub-cases. Recall the definition of a strong perfect elimination order from Chapter 2, this will be repeatedly used.

Case 2.2.1: $a \leq i$. By applying the strongly chordal property twice, we first establish that $v_j \in N[v_i]$ and then that $v_j \in N[v_t]$. Given that $a \leq i$, $i < j$, $v_a \in N[v_i]$, $v_i \in N[v_i]$, and $v_a \in N[v_j]$, the strongly chordal property ensures that $v_j \in N[v_i]$. Moreover, since $i \leq t$, $i < j$, $v_j \in N[v_i]$, $v_t \in N[v_i]$, and $v_i \in N[v_i]$, we conclude that $v_j \in N[v_t]$.

Case 2.2.2: $i < a$. The result follows directly from the strongly chordal property. Since $v_a \in N[v_i]$, we have $a \leq t$ because the algorithm selects $v_t \in N[v_i]$ to maximize the index t . Thus, given $a \leq t$, $i < j$, and the conditions $v_a \in N[v_i]$, $v_i \in N[v_t]$ and $v_j \in N[v_a]$, the strongly chordal property ensures that $v_j \in N[v_t]$. $\square$

To finish, observe that as a direct consequence of Claim 4.3.3 there is some minimum dominating set D_n^* such that $D_n \subseteq D_n^*$, which implies that D_n is also a minimum dominating set. Thus, Lemma 4.3.3 implies Theorem 4.3.1. $\square$

4.4 Sublinear $O(n \log n)$ algorithm

The bottom-up enumeration order of an RDV graph is a strong perfect elimination order, so Algorithm 3 will return a minimum dominating set when run on that order. A straightforward implementation of this algorithm will require $O(n + m)$ time. Can we speed this up to $O(n \log n)$? To do this, we need to show that the following two operations in each iteration of Algorithm 3 can be implemented in $O(\log n)$ time, at least when amortized over all iterations.

- A. Finding the vertex $v_t \in N[v_i]$ with highest index t in the bottom-up enumeration order.
- B. Marking all neighbors of v_t as “dominated”.

For both operations, we will use Lemma 3.4.1 along with a suitable data structure. Specifically, we utilize priority search trees [46] for Operation A and dynamic orthogonal ray shooting for Operation B. In particular, observe that Operation A is essentially the

same query we considered in Chapter 3, but requires us to find $v_t \in N[v_t]$ maximizing t , instead of $v_t \in N(v_i)$ minimizing t .

4.4.1 Implementing Operation A in $O(\log n)$ time per iteration

We want to identify the vertex $v_t \in N[v_i]$ that maximizes t in $O(\log n)$ amortized time. We store tuples in a priority search tree $\mathcal{P}$ according to the following invariant. Recall from Chapter 3 that we associate the value weight tuple $[x(b(v_t)), t]$ with vertex v_t .

Invariant 2: While processing vertex v_i , the priority search tree $\mathcal{P}$ should only store tuples of vertices v_t (for $t \geq i$) for which $y(b(v_t)) \geq y(t(v_i)) \geq y(t(v_t))$, i.e., all vertices including and ahead of v_i in the vertex order whose vertical segments encompass the current y -coordinate within their y -range.

Lemma 4.4.1 (Efficient Operation A). *Assuming a priority search tree is maintained according to Invariant 2, in Algorithm 3, one can find $v_t \in N[v_i]$ that maximizes the index t in $O(\log n)$ time.*

Proof. By Lemma 3.4.1, to find vertex $v_t \in N[v_i]$ that maximizes t , we can simply perform a range maximum query with the x -range of $\mathbf{h}(v_i)$. Invariant 2 ensures that y -coordinates of $\mathbf{h}(v_i)$ and $\mathbf{v}(v_t)$ line up, and the query range ensures that the x -coordinates do, which means that any tuple with value in the query range belongs to a vertex that is in $N_{\geq}[v_i]$. Among such vertices, the one with maximum weight t is the furthest one in the elimination order (as is required by Algorithm 3). Since we need to perform one range maximum query, this takes $O(\log n)$ time. $\square$

4.4.2 Implementing Operation B in $O(\log n)$ time per iteration

In Operation B, we need to mark all undominated vertices in $N[v_t]$ as dominated. Recall that $N_{<}[v_t]$ and $N_{>}[v_t]$ denote the vertices in $N[v_t]$ that come before and after v_t in the vertex ordering. The main idea is to maintain two data structures, D_H and D_V storing horizontal and vertical segments respectively corresponding to every undominated vertex. D_H is used to identify $N_{<}(v_t)$, while D_V is used to identify $N_{>}(v_t)$. As we will see, using Lemma 3.4.1, Operation B can then be viewed as a *range reporting* query within line segments. Additionally, a dynamic variant is required since we need to remove segments from

D_H and D_V once their corresponding vertices are dominated. This allows leveraging dynamic orthogonal ray shooting data structures to achieve an $O(n \log n)$ runtime (reviewed in Section 2.4, Chapter 2). Recall that these data structures maintain a dynamic set of horizontal line segments and support queries that return the first horizontal segment *hit* by a vertical query ray $\vec{q}$. By rotational symmetry, an analogous data structure can be used to maintain vertical segments while supporting horizontal ray queries.

Lemma 4.4.2 (Efficient Operation B). *Assuming suitable orthogonal ray shooting data structures are maintained, in Algorithm 3, one can implement Operation B, that is, mark all vertices in $N[v_t]$ as “dominated”, in amortized $O(\log n)$ time.*

Proof. At the start of the algorithm, we initialize two orthogonal ray shooting data structures D_H (storing horizontal segments) and D_V (storing vertical segments). For every vertex v of the graph, add $\mathbf{h}(v)$ to D_H and $\mathbf{v}(v)$ to D_V .

Whenever a vertex becomes dominated, we remove its associated segment from *both* D_H and D_V , maintaining the invariant that only segments corresponding to undominated vertices are stored in D_H and D_V at any time.

Due to Theorem 3.4.2, a vertex u is in $N[v_t]$ if and only if $\mathbf{h}(u) \cap \mathbf{v}(v_t) \neq \emptyset$ or $\mathbf{v}(u) \cap \mathbf{h}(v_t) \neq \emptyset$. The problem thus reduces to finding all (horizontal) segments in D_H that intersect the vertical segment $\mathbf{v}(v_t)$, and finding all (vertical) segments in D_V that intersect the horizontal segment $\mathbf{h}(v_t)$. Theorem 3.4.2 ensures that every vertex in $N[v_t]$ (and only those vertices) will be identified. Further, since we remove segments from both D_H and D_V as soon as a vertex is identified as dominated—regardless of whether it was identified via D_H or D_V —we ensure that no time is wasted searching among vertices that have already been marked as dominated.

To find these segments in D_H and D_V , we can simply perform repeated ray shooting queries by transforming the query line segment into a query ray.

Claim 4.4.3. *For any vertical query segment $q = \mathbf{v}(v_t)$, one can find all segments in D_H intersecting q in $O(k \log |D_H|)$ time where k is the number of segments returned. By rotational symmetry, an analogous result holds for horizontal query segments and D_V .*

Proof. We will use the orthogonal ray shooting data structure of Giyora and Kaplan [32], which was reviewed in Chapter 2. We first transform q into a ray, and then perform

repeated ray shooting queries. Replace the vertical segment q by a vertical ray $\vec{q}$ obtained by directing q upward. So, the ray originates at the lower end of q and goes vertically upward towards smaller y -coordinates.

We can iteratively find all segments intersecting q as follows. First, identify the first segment s which is hit by the ray $\vec{q}$, and then check whether the y -coordinate lies in the range defined by q . If so, remove s from D_H and the corresponding vertical segment from D_V , and repeat the query until no such segment is found. This process clearly ensures that all segments intersecting q are found.

If k is the number of segments returned, this process will require $O(k)$ deletions and $O(k)$ queries of the orthogonal ray shooting data structure. Thus, the total time needed is $O(k \log |D_H|)$. $\square$

Initializing D_H and D_V involves $2n$ insertions, so takes $O(n \log n)$ time. By Claim 4.4.3, in iteration i , it takes $O(k_{h,i} \log |D_H| + k_{v,i} \log |D_V|)$ time to find all vertices in $N[v_i]$ (here $k_{h,i}$ and $k_{v,i}$ are the number of vertices identified by querying D_H and D_V respectively). As $|D_H|, |D_V| \leq n$, this is $O((k_{h,i} + k_{v,i}) \log n)$. Thus in total, this takes $O(\sum_{i \in [n]} ((k_{h,i} + k_{v,i})) \log n)$ time. Since we always remove dominated vertices from both D_H and D_V immediately, we know $\sum_{i \in [n]} (k_{h,i} + k_{v,i}) = n$, so the total time is $O(n \log n)$ or $O(\log n)$ amortized time per iteration. $\square$

It is worth noting that the ray shooting data structure of Giyora and Kaplan, which uses dynamic fractional cascading and a generalized version of van Emde Boas trees, is quite complex. Instead, an older data structure of Overmars [56] directly supports the query needed in Claim 4.4.3 (find all horizontal segments intersecting a query vertical segment) and is considerably more simpler. However, Overmars' data structure only supports $O(\log^2 n)$ time insertions and deletions, and $O(k + \log^2 n)$ time queries, so we would incur an extra $\log n$ factor in the runtime bounds of Lemma 4.4.2.

4.4.3 Putting everything together, and outlook

We now put everything together in Algorithm 4. As earlier, we assume that the range maximum query or orthogonal ray shooting query returns a null value if no tuple exists within the specified range or if no segment is found. Also, for a segment s , let $s|_x$ and $s|_y$ denote the x/y -coordinate ranges of s (also known as the *projection* along an axis).

Theorem 4.4.4. *Given an n vertex graph G with an $O(n)$ sized RDV representation T , a minimum dominating set of G can be found in $O(n \log n)$ time.*

Proof. Parse T to compute the x -coordinates and y -coordinates of all nodes in T , then bucket-sort the vertices by decreasing $y(t(v))$ to obtain the bottom-up enumeration order $v_1, \dots, v_n$ in $O(n)$ time. Applying Algorithm 3 with this vertex-ordering will give a minimum dominating set, as guaranteed by Theorem 4.3.1. As discussed, the runtime of Algorithm 4 is $O(n)$ plus the time for executing Operation A and Operation B once per iteration.

By Lemma 4.4.1 and Lemma 4.4.2, both operations A and B can be implemented in amortized time $O(\log n)$ per iteration, which leads to an overall runtime of $O(n \log n)$. $\square$

We conclude with an open question. Can the runtime of Theorem 4.4.4 be improved to $O(n \log \log n)$ or even $O(n)$? One possible approach is to improve the runtime of priority search trees and orthogonal ray shooting data structures, at least in the special situation when all coordinates are integers in $O(n)$. However, it is also conceivable that a completely different method exists, akin to the $O(n)$ algorithm for interval graphs. Could a similar approach apply to RDV graphs as well?

Algorithm 4: Reformulated greedy algorithm for dominating set in RDV graphs

Input: An RDV graph G with a vertex order $v_1, \dots, v_n$

```
1 Initialize the dominating set  $D = \{\}$ 
2 Mark all vertices as “undominated”.
3 Initialize a priority search tree  $\mathcal{P}$ .
4 Initialize orthogonal ray shooting data structures  $D_H$  and  $D_V$ .
5 for  $i = 1, 2, \dots, n$  do
6    $D_H.insert(\mathbf{h}(v_i))$ 
7    $D_V.insert(\mathbf{v}(v_i))$ 
8 for  $i = 1, 2, \dots, n$  do
9   for all vertices  $v_t$  such that  $y(t(v_{i-1})) > y(b(v_t)) \geq y(t(v_i))$  do
10     $\mathcal{P}.insert([x(b(v_t)), t])$ 
11  if  $v_i$  is “undominated” then
12     $[\cdot, t] \leftarrow \mathcal{P}.rangemax[\mathbf{h}(v_i)|_x]$  // Operation A
13     $D \leftarrow D \cup \{v_t\}$ 
14     $\vec{q}_v \leftarrow$  upward ray from lower endpoint of  $\mathbf{v}(v_t)$  // Operation B
15     $s = D_H.orthrayshoot(\vec{q}_v)$ 
16    while  $s$  is not null and  $s|_y$  lies in  $\mathbf{v}(v_t)|_y$  do
17      Let  $u$  be the vertex with  $\mathbf{h}(u) = s$ 
18      Mark  $u$  as “dominated”
19       $D_H.delete(\mathbf{h}(u))$ 
20       $D_V.delete(\mathbf{v}(u))$ 
21       $s \leftarrow D_H.orthrayshoot(\vec{q}_v)$ 
22     $\vec{q}_h \leftarrow$  rightward ray from left endpoint of  $\mathbf{h}(v_t)$ 
23     $s = D_V.orthrayshoot(\vec{q}_h)$ 
24    while  $s$  is not null and  $s|_x$  lies in  $\mathbf{h}(v_t)|_x$  do
25      Let  $u$  be the vertex with  $\mathbf{v}(u) = s$ 
26      Mark  $u$  as “dominated”
27       $D_H.delete(\mathbf{h}(u))$ 
28       $D_V.delete(\mathbf{v}(u))$ 
29       $s \leftarrow D_V.orthrayshoot(\vec{q}_h)$ 
30   $\mathcal{P}.delete([x(b(v_i)), i])$ 
31 return  $D$ 
```

Chapter 5

Conclusion

The overarching goal of this thesis was to answer the following question: Several algorithmic problems have more efficient algorithms for interval graphs, as compared to general graphs. RDV graphs is a graph class which is a superclass of interval graphs. Given this, it is natural to investigate whether these algorithms can be extended to RDV graphs with the same runtime. Towards this goal, we provided $O(n \log n)$ algorithms for the maximum matching and minimum dominating set problem. In Section 3.5 and Section 4.4, we discussed some open problems related to improving the runtimes of these algorithms.

The essence of our approach is that in RDV graphs neighborhoods can be expressed via segment-sets, and any aggregate question about a neighborhood (such as finding minimum or maximum under some objective) becomes a query that answers this question among a set of segments. There are many other greedy algorithms (for other problems) that operate in this spirit, and we expect that for most of them, the run time could be reduced to $O(n \log n)$ time in RDV graphs using suitable data structures for segments. We demonstrate this for two problems (k -clique packing and maximum independent set) in Section 5.1 and Section 5.2. We then briefly discuss in Section 5.3 graphs with tree representations where subtrees have constant number of leaves (which is a generalization of RDV graphs), and whether results transfer.

5.1 Perfect Clique Packing in RDV graphs

In this section, we consider the *perfect k -clique packing* problem. Given a graph G and an integer k , we want to test whether there exists a partition $V_1, V_2, \dots, V_{\frac{n}{k}}$ of the vertex set V such that for every i , the graph induced by V_i is a k -clique, i.e., a complete graph on k vertices. For $k = 2$ this is exactly the question of finding a perfect matching. On the other hand, the problem is NP-Hard even for $k = 3$ [27]. Dahlhaus and Karpinski [19] studied perfect k -clique packing (which they called *perfect k -multidimensional matching*) and showed that a natural greedy strategy (see Algorithm 5) finds it (if one exists) for strongly chordal graphs.

Algorithm 5: Greedy-algorithm for perfect k -clique packing

Input: A graph G with a vertex order $v_1, \dots, v_n$

```

1 Initialize the packing  $M = \emptyset$ 
2 Initialize all vertices as “unmarked”
3 for  $i = 1, \dots, n$  do if  $v_i$  is “unmarked” then
4     if  $N_{>}(v_i)$  has  $k - 1$  “unmarked” vertices then
5         Let  $v_{j_1}, v_{j_2}, \dots, v_{j_{k-1}} \in N_{>}(v_i)$  be such that  $v(j_t)$  (for  $t \in [k - 1]$ ) is the vertex
            with  $t^{th}$  smallest index in  $N_{>}(v_i)$ 
6          $M \leftarrow M \cup \{(v_i, v_{j_1}, v_{j_2}, \dots, v_{j_{k-1}})\}$ 
7         “Mark”  $v_i, v_{j_1}, v_{j_2}, \dots, v_{j_{k-1}}$ 
8     else return “No perfect  $k$ -clique packing”
9 return  $M$ 

```

The correctness of this greedy algorithm is non-trivial and is omitted here, for a full proof see [19]. When implemented in a straightforward manner, this algorithm runs in $O(n + m)$ time. However, using our techniques in Chapter 3, we can easily reduce the runtime to $O(n \log n)$ in an RDV graph by performing $k - 1$ range minimum queries per vertex v_i . As earlier, we add and remove a vertex from the priority search tree at most once, and make n range minimum queries in total. Thus, the total runtime is $O(n \log n)$. For more details, see the pseudocode in Algorithm 6.

Algorithm 6: Reformulated Greedy-algorithm for perfect k -clique packing in RDV graphs

Input: An RDV graph G with a bottom-up enumeration order $v_1, \dots, v_n$

```

1 Initialize the packing  $M = \emptyset$ 
2 Initialize all vertices as “unmarked”
3 Initialize a priority search tree  $\mathcal{P}$ 
4 for  $i = 1, \dots, n$  do
5     for all vertices  $v_t$  such that  $y(t(v_{i-1})) > y(b(v_t)) \geq y(t(v_i))$  do
6          $\mathcal{P}.\text{insert}([x(b(v_t)), t])$ 
7     if  $v_i$  is “unmarked” then
8          $\mathcal{P}.\text{delete}([x(b(v_i)), i])$ 
9         for  $t = 1, 2, \dots, k - 1$  do
10             $[\cdot, j_t] \leftarrow \mathcal{P}.\text{rangemin}[\mathbf{h}(v_i)|_x]$ 
11             $\mathcal{P}.\text{delete}([x(b(v_{j_t})), j_t])$ 
12        if  $j_1, j_2, \dots, j_{k-1}$  are all not null then
13             $M \leftarrow M \cup \{(v_i, v_{j_1}, v_{j_2}, \dots, v_{j_{k-1}})\}$ 
14            “Mark”  $v_i, v_{j_1}, v_{j_2}, \dots, v_{j_{k-1}}$ 
15        else return “No perfect  $k$ -clique packing”
16 return  $M$ 

```

5.2 Maximum Independent Set in RDV graphs

In this section, we show that our techniques also give an $O(n \log n)$ algorithm for maximum independent set. An *independent set* is a subset $S \subseteq V$ of vertices in a graph such that no two vertices in S share an edge. The *maximum independent set* is an independent set with largest possible number of vertices. Finding the maximum independent set in general graphs is NP-Hard [27], but there is a simple linear time greedy algorithm due to Gavril [31], which returns the maximum independent set in chordal graphs. The pseudocode for this algorithm is given in Algorithm 7. The greedy idea is straightforward: Initialize the independent set to be empty, then process the vertices in the perfect elimination order $v_1, v_2, \dots, v_n$, and greedily add the current vertex to the independent set if you can.

Clearly, Algorithm 7 always returns an independent set, though not necessarily a max-

Algorithm 7: Greedy-algorithm for independent set

Input: A graph G with a vertex order $v_1, \dots, v_n$

```
1 Initialize the independent set  $I \leftarrow \emptyset$ 
2 Initialize all vertices to be “unmarked”
3 for  $i = 1, \dots, n$  do
4   if  $v_i$  is “unmarked” then
5     “Mark” all vertices in  $N_{\geq}[v_i]$ 
6      $I \leftarrow I \cup \{v_i\}$ 
7 return  $I$ 
```

imum independent set.

Lemma 5.2.1. *Algorithm 7 returns a maximum independent set, when run on the perfect elimination order of a chordal graph.*

Proof. Let $\mathcal{I} \subseteq [n]$ denote the set of indices of vertices returned by the algorithm. For all $i \in \mathcal{I}$, let S_i denote the set of vertices which are freshly “marked” after v_i is processed. Observe that for all $i \in [n]$, S_i is clique, since $v_1, v_2, \dots, v_n$ is a perfect elimination order. As every vertex is marked at the end, the collection $\{S_i | i \in \mathcal{I}\}$ is a *clique cover* of size $|\mathcal{I}|$. But in any graph with a clique cover of size k , the maximum independent set can have size at most k since we can pick at most one vertex from each clique to be in the independent set. Hence, I is a maximum independent set. $\square$

Algorithm 7 will take $O(m + n)$ time when implemented in the obvious way. Using our techniques from Chapter 3 and Chapter 4, it is straightforward to speed up Algorithm 7 to have $O(n \log n)$ runtime in an RDV graph.

Theorem 5.2.2. *Given an n vertex graph G with an $O(n)$ sized RDV representation T , a maximum independent set of G can be found in $O(n \log n)$ time.*

Proof. Parse T to compute the x -coordinates and y -coordinates of all nodes in T , then bucket-sort the vertices by decreasing $y(t(v))$ to obtain the bottom-up enumeration order $v_1, \dots, v_n$ in $O(n)$ time. Applying Algorithm 7 with this vertex-ordering will give a maximum independent set, as guaranteed by Lemma 5.2.1.

At the start, initialize a orthogonal ray shooting data structure D_V storing vertical segments corresponding to all vertices. Then, while processing vertex v_i , we can mark all vertices in $N_{\geq}[v_i]$ by searching for segments that intersect $\mathbf{h}(v_i)$ in D_V . This is same idea as Operation B in Chapter 4. Following the proof in Lemma 4.4.2, we get a total $O(n \log n)$ running time. $\square$

This leaves us with few possible lines of research. What about other graph problems? Two problems which comes to mind are vertex coloring and maximum clique. For both, there exists linear-time greedy algorithms that work for chordal graphs, but it is not clear how to use our techniques as is to obtain a runtime improvement for those algorithms.

5.3 Other graph classes

Recall that an RDV graph is a chordal graph with a rooted tree representation where every subtree $T(v)$ has exactly one leaf. A natural generalization of this graph class are the chordal graphs with a rooted tree representation where every subtree $T(v)$ has at most Δ leaves, for some constant Δ . A very similar concept was introduced by Chaplick and Stacho under the name of *vertex leafage* [12]; the only difference is that they considered unrooted host trees and so count the root of $T(v)$ as leaf if it has degree 1.

It is easy to check that a result similar to Lemma 3.4.1 holds for these graphs. For each vertex v , split $T(v)$ into $k \leq \Delta$ paths $P_1(v), \dots, P_k(v)$, each connecting the root $t(v)$ of $T(v)$ to a leaf, such that their union covers all of $T(v)$. Define segment $\mathbf{h}(v)$ as before (it only depends on $t(v)$), and define k vertical segments $\mathbf{v}_1(v), \dots, \mathbf{v}_k(v)$, one for each of the paths. One easily verifies that for $i < j$ vertex v_j is a neighbor of v_i if and only if at least one of $\mathbf{v}_1(v_j), \dots, \mathbf{v}_k(v_j)$ intersects $\mathbf{h}(v_i)$.

Can we use this to design fast algorithms for graph problems? There are two difficulties with this approach. First, the greedy algorithm for maximum matching and minimum dominating set only returns optimal solutions for strongly chordal graphs. Graphs with tree representations where subtrees have few leaves need not be strongly chordal, since even path graphs (discussed in Section 2.2.3) need not be strongly chordal. Thus, there is no guarantee that these algorithms will work, and a different approach is likely needed. For a concrete example, see Figure 5.1 for a path graph in which the greedy matching algorithm fails. Second, since there are multiple vertical segments now, we do not have a

clean result as in Theorem 3.4.2. This makes it hard to use orthogonal ray shooting in a similar way as in Chapter 4.

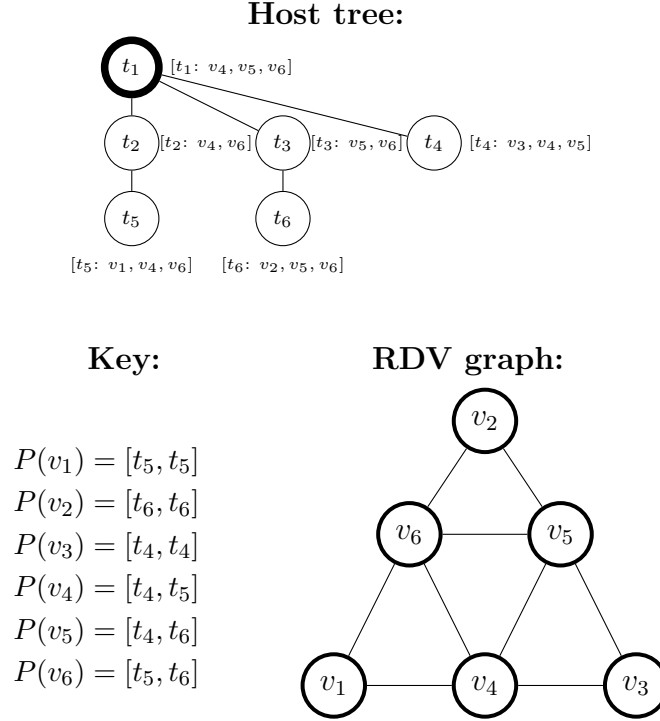


Figure 5.1: An example of a path graph: the 3-*trampoline* (labelled with v_i) and its rooted host tree (labelled with t_i). The path corresponding to vertex v_i is $P(v_i)$, and is indicated by the endpoint nodes of that path. Next to each node of the host tree, the paths that include it have been listed. The greedy matching algorithm, when run on the bottom-up enumeration $v_1, v_2 \dots, v_6$, will return the matching $\{(v_1, v_4), (v_2, v_5)\}$, even though the graph has a matching of size 3.

Nevertheless, one can still use priority search trees in a similar way as in Chapter 3, to speed up the greedy matching Algorithm 1 to a $O(n \log n)$ runtime. While this algorithm will not return a maximum matching, it will still return a 2-approximation to the maximum matching, so one can view this as a sublinear algorithm to find a 2-approximation. The algorithm will be similar to Algorithm 2, except that now each vertex has k vertical segments. We will maintain the same invariant for the vertical segments stored in the priority search tree, adding (and removing) a vertical segment at the appropriate time (note that

vertical segments corresponding to the same vertex might be added at different times). As before, the range minimum query will ensure that v_i is matched with the right vertex. All other aspects of the implementation are the same.

This leaves us with some natural open questions: Can we design optimal greedy algorithms for graphs with tree representations with bounded leaf subtrees? Conversely, can we identify “restrictions” on RDV graphs that keep them broader than interval graphs while allowing for better than $O(n \log n)$ runtimes for matching, dominating set, and independent set?

References

- [1] L. Babel, I.N. Ponomarenko, and G. Tinhofer. The isomorphism problem for directed path graphs and for rooted directed path graphs. *Journal of Algorithms*, 21(3):542–564, 1996.
- [2] G. Balakrishnan, S. Chakraborty, N.S. Narayanaswamy, and K. Sadakane. Succinct data structure for path graphs. *Information and Computation*, 296:105124, 2024.
- [3] C. Berge. *Graphs and Hypergraphs, 2nd edition*. North-Holland, 1976. Translated from *Graphes et Hypergraphes*, Dunod, 1970.
- [4] T. Biedl and P. Gokhale. Finding maximum matchings in RDV graphs efficiently. In *Proceedings of the 36th Canadian Conference on Computational Geometry (CCCG 2024) Brock University, St. Catharines, Canada*, pages 305–312, 2024.
- [5] Therese Biedl. Strongly chordal graphs as intersection graphs of trees (Farber’s proof revisited), 2025.
- [6] K. Booth and J. Howard Johnson. Dominating sets in chordal graphs. *SIAM Journal on Computing*, 11(1):191–199, 1982.
- [7] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Computing and System Sciences*, 13:335–379, 1976.
- [8] Kathie Cameron, Chính T. Hoàng, and Benjamin Lévêque. Characterizing directed path graphs by forbidden asteroid. *Journal of Graph Theory*, 68(2):103–112, 2011.
- [9] Manoel B. Campêlo and Sulamita Klein. Maximum vertex-weighted matching in strongly chordal graphs. *Discrete Applied Mathematics*, 84(1):71–77, 1998.

- [10] S. Chaplick. PQR-trees and undirected path graphs. Master’s thesis, Department of Computer Science, University of Toronto, 2008. <https://hdl.handle.net/1807/118597>.
- [11] S. Chaplick. Intersection graphs of non-crossing paths. *Discrete Mathematics*, 346(8):113498, 2023.
- [12] S. Chaplick and J. Stacho. The vertex leafage of chordal graphs. *Discrete Applied Mathematics*, 168:14–25, 2014.
- [13] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Commun. ACM*, 66(12):85–92, Nov 2023.
- [14] Julia Chuzhoy and Sanjeev Khanna. *A Faster Combinatorial Algorithm for Maximum Bipartite Matching*, pages 2185–2235.
- [15] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.
- [16] T. H. Cormen, C. E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd edition*. MIT Press, McGraw-Hill Book Company, 2009.
- [17] D.G. Corneil and Y. Perl. Clustering and domination in perfect graphs. *Discrete Applied Mathematics*, 9(1):27–39, 1984.
- [18] D.G. Corneil and Lorna K. Stewart. Dominating sets in perfect graphs. *Discrete Mathematics*, 86(1):145–164, 1990.
- [19] E. Dahlhaus and M. Karpinski. Matching and multidimensional matching in chordal and strongly chordal graphs. *Discrete Applied Mathematics*, 84(1):79–91, 1998.
- [20] Hubert de Fraysseix, Patrice Ossona de Mendez, and Pierre Rosenstiehl. On triangle contact graphs. *Comb. Probab. Comput.*, 3:233–246, 1994.
- [21] Paul F. Dietz, Merrick Furst, and John E. Hopcroft. A linear time algorithm for the generalized consecutive retrieval problem. Technical report, Cornell University, USA, 1979.

- [22] F. Bauernöppel and E. Kranakis and D. Krizanc and A. Maheshwari and J.-R. Sack and J. Urrutia. An improved maximum matching algorithm in a permutation graph, 1995. Manuscript. Improved version of Technical Report TR-95-06, School of Computer Science, Carleton University, Ottawa, Canada.
- [23] M. Farber. *Applications of L.P. duality to problems involving independence and domination*. PhD thesis, Rutgers University, 1982.
- [24] M. Farber. Characterizations of strongly chordal graphs. *Discret. Math.*, 43(2-3):173–189, 1983.
- [25] Martin Farber. Domination, independent domination, and duality in strongly chordal graphs. *Discrete Applied Mathematics*, 7(2):115–130, 1984.
- [26] Martin Farber and J. Mark Keil. Domination in permutation graphs. *Journal of Algorithms*, 6(3):309–321, 1985.
- [27] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
- [28] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.
- [29] F. Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discret. Math.*, 13(3):237–249, 1975.
- [30] F. Gavril. A recognition algorithm for the intersection graphs of paths in trees. *Discret. Math.*, 23(3):211–227, 1978.
- [31] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.
- [32] Y. Gijora and H. Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5(3):28:1–28:51, 2009.
- [33] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1st edition, 1980.

- [34] M.C. Golumbic, D. Rotem, and J. Urrutia. Comparability graphs and intersection graphs. *Discrete Mathematics*, 43(1):37–46, 1983.
- [35] Daniel Gonçalves, Lucas Isenmann, and Claire Pennarun. *Planar Graphs as L-intersection or L-contact graphs*, pages 172–184.
- [36] 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.
- [37] Hamed Hatami and Pooya Hatami. The Implicit Graph Conjecture is False . In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1134–1137, Los Alamitos, CA, USA, November 2022. IEEE Computer Society.
- [38] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 334–343, New York, NY, USA, 1988. Association for Computing Machinery.
- [39] H. Kaplan, E. Molad, and R. Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 639–648, New York, NY, USA, 2003. Association for Computing Machinery.
- [40] Boland-J. Lekkerkerker, C. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51(1):45–64, 1962.
- [41] Y. Liang and C. Rhee. Finding a maximum matching in a circular-arc graph. *Information Processing Letters*, 45(4):185–190, 1993.
- [42] M.-S. Lin and S.-H. Su. Counting maximal independent sets in directed path graphs. *Information Processing Letters*, 114(10):568–572, 2014.
- [43] M.-S. Lin and C.-C. Ting. Computing the k-terminal reliability of directed path graphs. *Information Processing Letters*, 115(10):773–778, 2015.
- [44] Min Chih Lin and Jayme Luiz Szwarcfiter. Characterizations and recognition of circular-arc graphs and subclasses: A survey. *Discret. Math.*, 309(18):5618–5635, 2009.

- [45] L. Lovász and M. D. Plummer. *Matching theory*, volume 29 of *Annals of Discrete Mathematics*. North-Holland Publishing Co., Amsterdam, 1986.
- [46] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
- [47] Terry A. McKee and F. R. McMorris. *Topics in intersection graph theory*. Society for Industrial and Applied Mathematics, January 1999.
- [48] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(1–4):215–241, 2023.
- [49] G. Mertzios, A. Nichterlein, and R. Niedermeier. A linear-time algorithm for maximum-cardinality matching on cocomparability graphs. *SIAM J. Discret. Math.*, 32(4):2820–2835, 2018.
- [50] S. Micali and V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27, 1980.
- [51] 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.
- [52] C. Monma and V. Wei. Intersection graphs of paths in a tree. *J. Comb. Theory, Ser. B*, 41(2):141–181, 1986.
- [53] John Harold Muller. *Local structure in graph classes*. PhD thesis, USA, 1988. Order No: GAX88-11342.
- [54] G. Narasimhan. A note on the hamiltonian circuit problem on directed path graphs. *Information Processing Letters*, 32(4):167–170, 1989.
- [55] Yakov Nekrich. Dynamic planar point location in optimal time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2021*, page 1003–1014, New York, NY, USA, 2021. Association for Computing Machinery.

- [56] Mark H. Overmars. Range searching in a set of line segments. In *Proceedings of the First Annual Symposium on Computational Geometry*, SCG '85, page 177–185, New York, NY, USA, 1985. Association for Computing Machinery.
- [57] B. S. Panda. The forbidden subgraph characterization of directed vertex graphs. *Discrete Mathematics*, 196(1):239–256, 1999.
- [58] C. Papadopoulos and S. Tzimas. Computing a minimum subset feedback vertex set on chordal graphs parameterized by leafage. *Algorithmica*, 86(3):874–906, 2024.
- [59] Peter Renz. Intersection representations of graphs by arcs. *Pacific J. Math.*, 34(2):501–510, August 1970.
- [60] C. Rhee and Y. Liang. Finding a maximum matching in a permutation graph. *Acta Informatica*, 32(8):779–792, 1995.
- [61] R. P. Swaminathan and Donald K. Wagner. The arborescence-realization problem. *Discrete Applied Mathematics*, 59(3):267–283, 1995.
- [62] Nicolas Trotignon. Perfect graphs: a survey, 2015.
- [63] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10:99–127, 1976.
- [64] Vijay V. Vazirani. A theory of alternating paths and blossoms from the perspective of minimum length. *Math. Oper. Res.*, 49(3):2009–2047, August 2024.
- [65] S. W. Cheng, M. Kaminski, and S. Zaks. Minimum dominating sets of intervals on lines. *Algorithmica*, 20(3):294–308, Mar 1998.

Appendix A

Supplementary Code

We present a brief code snippet (Python 3.13.3) to verify that the graph in Figure 2.3 is strongly chordal. Based on the characterization provided by Farber [24], it is sufficient to check that every even-length cycle with a length of at least 6 contains an odd chord. A list of all such cycles, along with one odd chord, is provided at the end.

```
1 import networkx as nx
2
3 # Create an empty graph
4 G = nx.Graph()
5
6 # List the edges
7 edges = [
8     ("v_1", "v_4"), ("v_1", "v_6"), ("v_4", "v_6"),
9     ("v_2", "v_4"), ("v_3", "v_6"), ("v_4", "v_5"),
10    ("v_6", "v_7"), ("v_4", "v_8"), ("v_6", "v_8"),
11    ("v_4", "v_7"), ("v_5", "v_6"), ("v_2", "v_5"),
12    ("v_5", "v_8"), ("v_7", "v_8"), ("v_3", "v_7")
13 ]
14
15 # Add edges to the graph
16 G.add_edges_from(edges)
17
```

```

18 # function to print each even cycle, and an odd chord for every
    cycle if one exists
19 def even_cycle_check(G):
20     for cycle in nx.simple_cycles(G):
21         l = len(cycle)
22         if (l>=6) and (l%2==0):
23             odd_chord = ()
24             for i in range(l):
25                 for j in range(l):
26                     if (j - i) % 2 == 1 and ((j - i) % l) != 1 and
                        ((j - i) % l) != l-1:
27                         if (cycle[i], cycle[j]) in G.edges():
28                             odd_chord = (cycle[i], cycle[j])
29             if odd_chord == ():
30                 print("Even Cycle:", cycle, "No Odd Chord Found \n
                        ")
31             else:
32                 print("Even Cycle:", cycle, "Odd Chord:",
                        odd_chord, '\n')
33
34 even_cycle_check(G)

```

Even Cycle	Odd Chord
$\{v_1, v_4, v_2, v_5, v_8, v_6\}$	(v_8, v_4)
$\{v_1, v_4, v_2, v_5, v_8, v_7, v_3, v_6\}$	(v_6, v_8)
$\{v_1, v_4, v_5, v_8, v_7, v_6\}$	(v_6, v_5)
$\{v_1, v_4, v_8, v_7, v_3, v_6\}$	(v_6, v_8)
$\{v_1, v_4, v_7, v_8, v_5, v_6\}$	(v_6, v_7)
$\{v_4, v_6, v_3, v_7, v_8, v_5\}$	(v_8, v_6)
$\{v_4, v_6, v_7, v_8, v_5, v_2\}$	(v_5, v_6)
$\{v_4, v_2, v_5, v_6, v_3, v_7\}$	(v_6, v_4)
$\{v_4, v_2, v_5, v_6, v_7, v_8\}$	(v_8, v_5)
$\{v_4, v_2, v_5, v_6, v_8, v_7\}$	(v_6, v_4)
$\{v_4, v_2, v_5, v_8, v_6, v_7\}$	(v_8, v_4)
$\{v_4, v_5, v_6, v_3, v_7, v_8\}$	(v_8, v_6)
$\{v_4, v_5, v_8, v_6, v_3, v_7\}$	(v_7, v_8)
$\{v_4, v_8, v_5, v_6, v_3, v_7\}$	(v_6, v_4)

Table A.1: List of even cycles and their corresponding odd chords.