Finding maximum matchings in RDV graphs efficiently

Therese Biedl
David R. Cheriton School of Computer Science, University of Waterloo

Prashant Gokhale
David R. Cheriton School of Computer Science, University of Waterloo

Abstract

In this paper, we study the maximum matching problem in *RDV graphs*, i.e., **v**ertex-intersection graphs of **d**ownward paths in a **r**ooted tree. We show that this problem can be reduced to a problem of testing (repeatedly) whether a horizontal segment intersects one of a dynamically changing set of vertical segments, which in turn reduces to a range minimum query. Using a suitable data structure, we can therefore find a maximum matching in $O(n \log n)$ time (presuming a linear-sized representation of the graph is given), i.e., without even looking at all edges.

Keywords and phrases maximum matching, RDV graphs, strongly chordal graphs

Digital Object Identifier 10.57717/cgt.v5i2.72

Related Version Preliminary version appeared at CCCG 2024 [3]. The results also appeared as part of the second author's Master's thesis [16].

Acknowledgements Research of TB supported by NSERC, RGPIN-2020-03958. Research of PG supported by a MITACS Globalink Graduate Fellowship.

1 Introduction

The MATCHING problem 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 [2, 24]. The fastest known algorithm for general graphs runs in $O(\sqrt{n}m)$ time ([27], see also [34]). There have been some recent break-throughs for algorithms for maximum flow [9, 19, 8] based on the Laplacian paradigm introduced by Spielman and Teng [32], culminating in an almost-linear run-time $O(m^{1+o(1)})$ algorithm for undirected graphs [8]. This immediately implies an almost-linear algorithm for MATCHING in bipartite graphs.

Greedy-algorithm and interval graphs. Naturally one wonders whether truly linear-time algorithms (i.e., with $O(m + n)$ run-time) exist for MATCHING, at least if the graphs have special properties. One natural approach for this is to use the greedy-algorithm for MATCHING shown in Algorithm 1, which clearly takes linear time. With a suitable vertex order this will always find the maximum matching (enumerate the vertices so that matched ones appear consecutively at the beginning); the challenge is hence to find a vertex order (without knowing the maximum matching) for which the greedy-algorithm is guaranteed to work.

One graph class where this can be done is the *interval graphs*, i.e., the intersection graphs of intervals on a straight line. It was shown by Moitra and Johnson [28] that the greedy-algorithm always finds a maximum matching in an interval graph as long as we sort the vertices by left endpoint of their intervals. This gives an $O(m + n)$ algorithm for interval graphs since an interval representation can be found in $O(m + n)$ time [4]. Liang and Rhee [20] improve this further to $O(n \log n)$ time (assuming the interval representation is given) by

Algorithm 1 Greedy-algorithm for matching.

Input: A graph G with a vertex order $v_1, \dots, v_n$
initialize the matching $M = \emptyset$ and mark all vertices as "unmatched"
for $i = 1, \dots, n$ **do**
  if v_i *is "unmatched" and has unmatched neighbours* **then**
    among all unmatched neighbours of v_i, let v_j be the one that minimizes j
    `// j > i, for otherwise` v_j `would have been matched earlier`
    add (v_i, v_j) to M and mark v_i and v_j as "matched"
return M

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 [33], as observed later by Liang and Rhee [31].

Our results. In this paper, we take inspiration from [20] and develop sub-linear algorithms for Matching in *RDV graphs*, i.e., graphs that can be represented as vertex-intersection graphs of downward paths in a rooted tree T. (This is called an *RDV representation*; formal definitions will be given in Section 2.) RDV graphs were introduced by Gavril [15]; many properties have been discovered and for many problems efficient algorithms have been found for RDV graphs [1, 21, 22, 30], quite frequently in contrast to only slightly bigger graph classes where the problem turns out to be hard. It is easy to see that all interval graphs are RDV graphs, so our results re-prove the $O(n \log n)$ run-time for interval graphs from [20] (sadly, the $O(n \log \log n)$ run-time from [31] does not seem to carry over to RDV graphs).

RDV graphs can be recognized in polynomial time, and along the way an RDV representation is produced [15]. (The run-time has been improved, and even a linear-time algorithm has been claimed but without published details; see [6, Section 2.1.4] for more on the history.)

We show in this paper that if we are given an n-vertex graph G with an RDV representation on a tree T, then we can find a maximum matching in $O(|T| + n \log n)$ time. There always exists an RDV representation of G with $|T| \in O(n)$, so if we are given a suitable one then the run-time becomes $O(n \log n)$, hence sub-linear.

Our idea is to use the greedy-algorithm (Algorithm 1), and to speed up the time it takes to find a minimum (in the vertex ordering) unmatched neighbour to $O(\log n)$ per query. The key ingredient here is that 'is v_i a neighbour 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). This turns the question of finding v_i's minimum (in the vertex order) neighbour into a range minimum query. Such queries can be performed with the help of priority search trees using linear space and $O(\log n)$ time per operation [25]; this gives our result since we need $O(n)$ operations. We use range minimum queries as a black box, so if the run-time were improved (e.g. one could dream of $O(\log \log n)$ run-time if coordinates are integers in $O(n)$, as they are in our application) then the run-time of our matching-algorithm would likewise improve.

While we focused on matching here, the 'obvious' linear-time greedy-algorithm works (in RDV graphs) also for perfect k-clique packing, which is a generalization of PerfectMatching, i.e., the problem of finding a matching of size $n/2$. We briefly discuss in Section 4.2 how this greedy-algorithm likewise can be implemented in $O(n \log n)$ time for RDV graphs.

Other related results: There are a number of other results concerning fast algorithms to solve MATCHING in intersection graphs of some geometric objects, see Table 1. Specifically, the results for interval graphs were extended to *circular arc graphs* (intersection graphs of arcs of a circle) [20]. In an entirely different approach, MATCHING can also be solved very efficiently in *permutation graphs* (intersection graphs of line segments connecting two parallel lines) [31]; see also [11] 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 [17]. For these, maximum matchings can be found in linear time [26].

We should note that RDV graphs are unrelated to circular arc graphs, permutation graphs and co-comparability graphs (i.e., neither a subclass nor a superclass); Figure 1 gives a specific example. As such, these results do not directly impact ours or vice versa.

Finally, the greedy-algorithm actually works for broader graph classes; in particular Dahlhaus and Karpinski [10] showed that it finds the maximum matching for *strongly chordal graphs*. These are the graphs that are *chordal* (every cycle C of length at least 4 has a *chord*, i.e., an edge between two non-consecutive vertices of C), and where additionally every even-length cycle C of length at least 6 has a chord (v, w) such that an odd number of edges of C lie between v and w. Every RDV graph is known to be strongly chordal (see Lemma 12 for a short proof).

The question whether chordal graphs have a linear-time algorithm for MATCHING remains open. But likely the answer is no, because as argued in [10], a linear-time algorithm for MATCHING (in particular therefore PERFECTMATCHING) would imply a linear-time algorithm for PERFECTMATCHING in any bipartite graph that is *dense* (has $\Theta(n^2)$ edges).

Graph Class	Runtime	Reference
Interval graphs	$O(n \log\log n)$	[20, 31]
Circular arc graphs	$O(n \log n)$	[20]
Permutation graphs	$O(n \log\log n)$	[31]
Strongly chordal graphs	$O(n + m)$	[10]
Co-comparability graphs	$O(n + m)$	[26]
RDV graphs	$O(n \log n)$	Section 3

Table 1 Existing and new results for MATCHING in some classes of graphs, presuming a suitable intersection representation is given and sufficiently small.

Our paper is structured as follows. After reviewing some background in Section 2, we give our main result for RDV graphs in Section 3. We briefly discuss extensions in Section 4.

2 Background

In this paper we study vertex-intersection graphs of subtrees of trees. We first define this formally, and then restrict the attention to a specific subclass.

▶ **Definition 1.** *Let G be a graph. A* representation of G as a vertex-intersection graph of subtrees of a tree *(or* tree-intersection representation *for short) consists of a* host-tree T *and, for each vertex v of G, a subtree $T(v)$ of T such that (v, w) is an edge of G if and only if $T(v)$ and $T(w)$ share at least one node of T.*

As convention, we use the term 'node' for the vertices of the host tree, to distinguish them from the vertices of the graph represented by it. A tree-intersection representation is

sometimes also called a *clique-tree* [29] or *characteristic tree* [15]. It is well-known that a graph has such an intersection representation if and only if it is chordal [14]. We now review some properties of tree-intersection representations where the host-tree has been rooted.

▶ **Definition 2.** *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) node 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 1 are enumerated according to a bottom-up enumeration.

It will be convenient to assign points to the nodes of the host-tree T as follows. First, fix an arbitrary order of children at each node, and then enumerate the leaves of T as $L_1, \ldots, L_\ell$ from left to right. For every node ν in T, let $\ell(\nu)$ be the leftmost (i.e., lowest-indexed) leaf that is a descendant of ν. Formally, if ν is a leaf, then set $\ell(\nu) = \nu$; otherwise set $\ell(\nu) = \ell(c)$ where c is the leftmost child of ν. Set $x(\nu)$ to be the index of $\ell(\nu)$. Symmetrically define $r(\nu)$ to be the rightmost leaf that is a descendant of ν. Also, define $y(\nu)$ to be the distance of node ν from the root of the host-tree. Figure 1 shows each node ν drawn at point $(x(\nu), y(\nu))$ (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. Note that a bottom-up enumeration of the vertices is the same as sorting the vertices by $y(t(\cdot))$; it therefore can be computed in $O(|T| + n)$ time, presuming every vertex v stores a reference to $t(v)$.

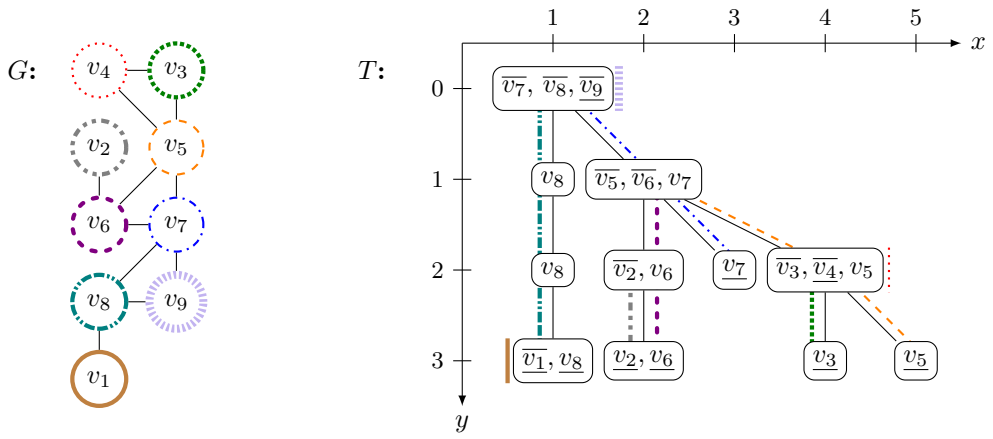

Figure 1 An RDV graph G together with one possible RDV representation with host-tree T (for illustrative purposes the representation is more complicated than needed). Each node ν lists those vertices v with $\nu \in T(v)$; we write $\underline{v}$ if $\nu = b(v)$ and $\overline{v}$ if $\nu = t(v)$. We also show the paths as poly-lines, with colors/dash-pattern matching the vertices. Nodes are drawn at their coordinates, and vertices are enumerated in bottom-up enumeration order. The graph is neither a circular arc graph nor a permutation graph (and hence also not a co-comparability graph), see Appendix A.

RDV graphs and friends: Numerous subclasses of chordal graphs can be defined by studying graphs that have a a tree-intersection representation where the subtrees or the host-tree have particular properties. Most prominent here is the idea to require that $T(v)$ is a path. This gives the *path graphs* (also known as *VPT graphs*). One can further restrict the paths to be directed (after imposing some edge-directions onto the host-tree); these are the *directed*

path graphs. One can restrict this even further by requiring that the edge-directions of the host-tree are obtained by rooting the host-tree, and this is the graph class that we study.

▶ **Definition 3.** *A* rooted directed path graph *(or* RDV graph *[29]) is a graph that has an* RDV representation*, i.e., a tree-intersection representation with a rooted host-tree where for every vertex v the subtree $T(v)$ is a* downward path*, i.e., a path that begins at some node and then always goes downwards.*

See Figure 1 for an example of an RDV representation. As a historical note, we want to mention that Gavril used 'directed path graphs' for RDV graphs [15] and only later papers (e.g. [1, 29]) distinguished further by whether the directions for the host-tree had to be obtained via rooting or could be chosen arbitrarily.

3 Matching in RDV graphs

Assume for the rest of this section that we are given an RDV representation of a graph G. In what follows, we will often use '$P(v)$' in place of '$T(v)$' for the subtree of a vertex v, to help us remember that these are downward paths rather than arbitrary trees. Recall that $t(v)$ denotes the top (closest to the root) node of $P(v)$; because we have a downward path (rather than an arbitrary tree) representing v we can now also define $b(v)$ to be the bottom node of $P(v)$. For run-time purposes we presume that 'the RDV representation is given' means that we have a rooted tree T and (for each vertex v of G) two references $b(v)$ and $t(v)$ to the nodes of T that define the downward path.

It follows easily from an intersection-representation result by Farber [12, Theorem 3.11] that every RDV graph is strongly chordal. Dahlhaus and Karpinski [10] showed that the greedy-algorithm works correctly on strongly chordal graphs if we consider vertices in a so-called *strong elimination order* (which is usually assumed to be given with a strongly chordal graph). This suggests that the greedy-algorithm works for RDV graphs, but there is one missing piece: How do we get a strong elimination order from an RDV representation efficiently? This is very easy (use the bottom-up enumeration), and the proof that it works is not hard and was likely known before; since we have not been able to find a reference for this we provide a proof in Appendix B.

▶ **Theorem 4.** *Let G be a graph with a given RDV representation. Then the greedy matching algorithm, applied to a bottom-up enumeration, returns a maximum matching.*

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 definitions first.

▶ **Definition 5.** *Let G be a graph with an RDV representation. For each vertex v, define the following (see Figure 2 for examples):*

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

Recall that $t(v)$ has the same x-coordinate as its leftmost descendant, so the x-range of segment $\mathbf{h}(v)$ is exactly the range of x-coordinates among descendants of $t(v)$.

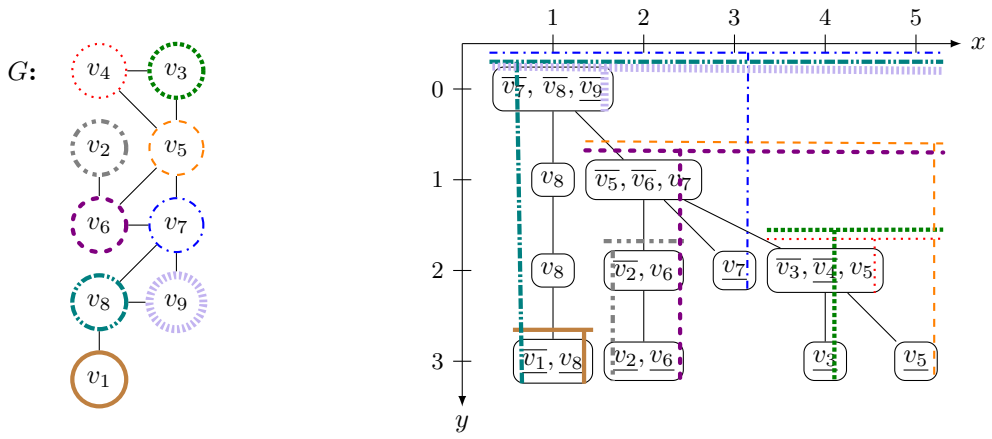

Figure 2 Mapping the vertices of the example in Figure 1 to horizontal and vertical segments. For ease of reading, segments are offset slightly, and degenerate segments (i.e., points) are shown as short segments.

▶ **Theorem 6.** *Let G be a graph with an RDV representation and let $v_1, \ldots, v_n$ be a bottom-up enumeration of vertices. 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 $u = t(v_i)$, 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 x-range of horizontal segment $\mathbf{h}(v_i)$ covers all such descendants.

Assume now that the segments intersect. By $y(t(v_j)) \leq y(t(v_i)) \leq y(b(v_j))$ then 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. ◀

In light of this insight, we can reformulate the greedy-algorithm and speed it up by using *range minimum queries*. In such a query, we are storing tuples $[x, w]$ (we call their aspects *value* and *weight*, respectively), we are given a range $[x', x'']$, and we want to find, among all tuples whose value falls into the given range, the one that minimizes the weight. We also want to dynamically add or remove tuples. McCreight [25] showed how to store the tuples in a so-called *priority search tree* such that insertion, deletion and range minimum queries can be done in $O(\log n)$ time for n currently stored tuples.

In our specific application, we associate each vertex v_i with a tuple that has value $x(b(v_i))$ and weight i. We maintain a priority search tree $\mathcal{P}$ with the invariant that, at the time when

processing vertex v_i (with current y-coordinate $y_{curr} := y(t(v_i))$), we store in $\mathcal{P}$ the tuples of those unmatched vertices v_j with $j > i$ where $y(b(v_j)) \geq y(t(v_i))$. Since we proceed in bottom-up enumeration order, $j > i$ implies $y(t(v_i)) \geq y(t(v_j))$, i.e., $\mathcal{P}$ contains only tuples of unmatched vertices whose vertical segments contain y_{curr} in their y-range. This means that to test whether v_i has a suitable neighbour, we can simply perform a range minimum query with the x-range of $\mathbf{h}(v_i)$, i.e., find the tuple whose value (i.e., the x-coordinate of some vertical segment $\mathbf{v}(v_j)$) falls into the x-range and that minimizes the weight (i.e., the index j). The invariant maintained for $\mathcal{P}$ ensures that y-coordinates line up, and the query range ensures that the x-coordinates do, which means that v_j is adjacent to v_i. Among such vertices, the one with minimum weight (i.e., index) is the smallest one in the elimination order as is required by the greedy-algorithm. Algorithm 2 shows how to implement this idea, and Figure 3 illustrates how $\mathcal{P}$ is used.

Algorithm 2 Reformulated greedy-algorithm for matching in RDV graphs

```
   Input: An RDV graph G with a bottom-up enumeration order v_1, ..., v_n
 1 initialize the matching M = ∅ and mark all vertices as "unmatched"
 2 initialize an empty priority search tree P
 3 y_curr = ∞                                          // current y-coordinate
 4 for i = 1, ..., n do
 5     y_prev ← y_curr; y_curr ← y(t(v_i))
 6     if (y_curr < y_prev) then                  // y_curr decreased, so update P
 7         for all vertices v_q such that y_prev > y(b(v_q)) ≥ y_curr do
 8             insert the tuple [x(b(v_q)), q] of v_q into P
 9     if v_i is "unmatched" then
10         delete the tuple of v_i from P
11         perform a range-minimum query in P w.r.t. to the x-range of h(v_i)
12         if the result of the range minimum query is not null, say it is [·, j] then
13             add (v_i, v_j) to M and mark v_i and v_j as "matched"
14             delete the tuple of v_j from P
15 return M
```

▶ **Lemma 7.** *Algorithm 2 returns a maximum matching and finishes in $O(n \log n)$ time.*

Proof. Observe first that the current y-coordinate can never increase, since it corresponds to the y-coordinate of $t(v_i)$, and the bottom-up enumeration sorts by exactly that. We add the tuple of v_q to $\mathcal{P}$ once the current y-coordinate reaches the bottom of the vertical segment of v_q; this happens exactly once for each vertex. Since we only match vertices that were already in $\mathcal{P}$, vertices newly added to $\mathcal{P}$ are hence unmatched, and we remove vertices from $\mathcal{P}$ once they are matched. So vertices with tuples in $\mathcal{P}$ are unmatched throughout, and they have index at least i since we remove the tuple of v_i from $\mathcal{P}$ when processing the vertex.

Next we argue that (v_i, v_j) is actually an edge if we reach line 12. Since v_j's tuple was in $\mathcal{P}$, the current y-coordinate had reached the bottom of $\mathbf{v}(v_j)$, i.e., $y_{curr} \leq y(b(v_j))$. On the other hand, $y_{curr} \geq y(t(v_j))$, for otherwise (by the bottom-up enumeration order) we would have $j < i$, impossible. Therefore whenever we reach line 12 in the algorithm, the y-coordinates of $\mathbf{h}(v_i)$ and $\mathbf{v}(v_j)$ line up, and the query ensures that their x-coordinates line up, so edge (v_i, v_j) exists by Theorem 6. Further, the query returns, among all possible

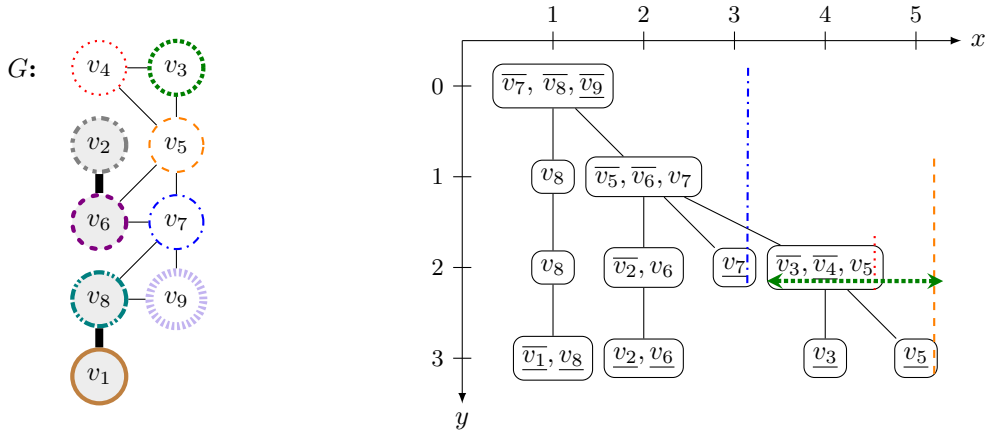

Figure 3 The query performed when processing v_3 (i.e., $i = 3$). We had earlier matched (v_1, v_8) and (v_2, v_6), so $\mathbf{v}(v_6)$ and $\mathbf{v}(v_8)$ are not in $\mathcal{P}$. Also $\mathbf{v}(v_9)$ is not in $\mathcal{P}$ since its y-range does not intersect $y_{curr} = 2$. We perform a query in the x-range of $\mathbf{h}(v_3)$, which contains the vertical segments of v_4 and v_5 (but not of v_7); the query returns the tuple of v_4.

vertices, the one with minimum weight, that is one which comes earliest in the order. Hence, we exactly implement the greedy-algorithm, and hence return a maximum matching.

For the runtime, observe that we perform $O(1)$ queries per iteration, and each query takes $O(\log n)$ time. Since there are at most n iterations, we will finish in $O(n \log n)$ time. Adding tuples in line 7 can easily be done in $O(n \log n)$ total time by precomputing a list of vertices sorted by the $y(b(v))$'s and then advancing through it in line 7; this takes constant time per vertex (which is either added or discarded since it was already matched). ◀

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

▶ **Theorem 8.** *Given an n-vertex graph G with an RDV representation with host-tree T, the maximum matching of G can be found in $O(|T| + n \log n)$ 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 $v_1, \ldots, v_n$ in $O(|T| + n)$ time. By Theorem 4 applying the greedy-algorithm with this vertex-ordering will give a maximum matching. Using Algorithm 2, the run-time is in $O(n \log n)$ as argued in Lemma 7. ◀

One can easily argue that any RDV graph has an RDV representation T with $|T| \in O(n)$, for 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. So the run-time becomes $O(n \log n)$ if a suitably small RDV representation is given.

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 [31]. This naturally raises an open question: Could the run-time of Theorem 8 also be improved to $O(n \log \log n)$ time, presuming $|T| \in O(n)$? 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. There are some improvements of the run-time for range minimum queries (and more generally, path-minimum queries): Brodding, Davoodi and Rao show how to do this in $O(\log n / \log \log n)$ time per operation in the RAM model [5]. However, their model is such that every value is used in at most one tuple, and it is not immediately obvious whether their data structures

could be used in our situation where one value (x-coordinate) could be mapped to multiple weights (indices of vertices). Can we achieve run-time $O(n \log n / \log\log n)$ for Matching in RDV graphs?

4 Extensions

4.1 Tree-intersection representations where subtrees have few leaves

An RDV graph is a chordal graph with a tree-intersection representation where the host-tree is rooted and every subtree $T(v)$ has exactly one leaf. A natural generalization of this graph class are the chordal graphs with a tree-intersection representation where every subtree $T(v)$ has at most Δ leaves. (A very similar concept was introduced by Chaplick and Stacho under the name of *vertex leafage* [7]; 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.)

▶ **Theorem 9.** *Let G be a graph with a tree-intersection representation where the host-tree T is rooted and all subtrees have at most Δ leaves. Then the greedy-algorithm applied to the bottom-up enumeration can be implemented in $O(|T| + \Delta n \log n)$ time.*

Proof. For each vertex v, split $T(v)$ into $k \leq \Delta$ paths $P_1(v), \ldots, P_k(v)$, each connecting the root $t(v)$ of $T(v)$ to a leaf of $T(v)$, 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 $\mathbf{v}_1(v), \ldots, \mathbf{v}_k(v)$, one for each of the paths. One easily verifies that for $i < j$ vertex v_j is a neighbour of v_i if and only if at least one of $\mathbf{v}_1(v_j), \ldots, \mathbf{v}_k(v_j)$ intersects $\mathbf{h}(v_i)$. So we add (or remove) k different tuples in $\mathcal{P}$ corresponding to the vertex v_j, instead of just one tuple. All other aspects of the greedy-algorithm are exactly as in Section 3. ◀

Unfortunately, this does not improve the time to find maximum matchings for such graphs, because there is no guarantee that the greedy-algorithm finds a maximum matching when applied with a bottom-up enumeration. To see a specific example, consider the *directed path graphs* (recall that these are obtained by requiring $T(v)$ to be a directed path after making the host-tree directed, but the edge-directions need not come from rooting the host-tree). This is a strict superclass of RDV graphs, for example the graph in Figure 4, which is also known as *4-trampoline*, is a directed path graph but not an RDV graph since it is not even strongly chordal [13]. For any choice of root, every path $T(v)$ becomes a subtree with at most two leaves, and so the greedy-algorithm can be implemented in $O(n \log n)$ time (presuming the host-tree was small). Unfortunately, this does not necessarily give a maximum matching, see Figure 4.

This raises another natural open problem: Can we find a maximum matching in a directed path graph quickly? Likely the answer is no, because inspection of the argument in [10] (that a linear-time algorithm for Matching implies a linear-time algorithm for PerfectMatching in any dense bipartite graph) shows that the construction is actually a directed path graph.

4.2 Extension to other problem

There are many other natural graph problems where an easy greedy-algorithm exists that finds the correct answer for some graph classes such as interval graphs or generalizations thereof. Naturally one wonders whether the run-time of such greedy-algorithms could be made sublinear for RDV graphs with the techniques that we have seen earlier. We will show that this is indeed the case for the perfect k-clique packing problem (defined below). We

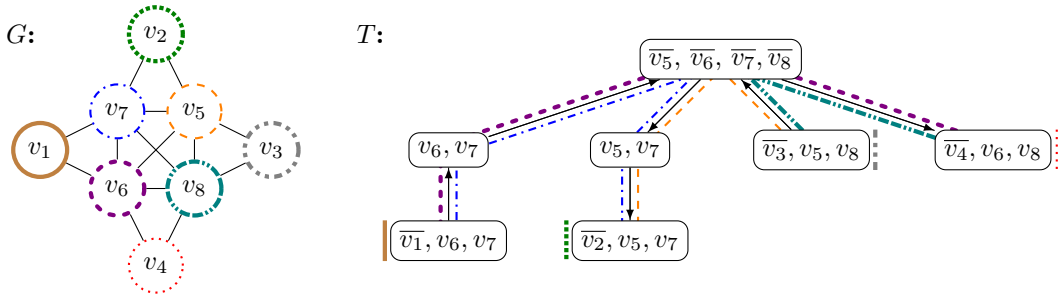

Figure 4 A directed path graph that is not strongly chordal. Correspondence of paths to vertices is shown via colors/dash-style, and one possible bottom-up enumeration is given. With this bottom-up enumeration, the greedy-algorithm would choose matching (v_1, v_6), (v_2, v_5), (v_3, v_8) and leave v_4 and v_7 unmatched even though the graph has a perfect matching.

have also been able to develop sublinear greedy-algorithms for the dominating set problem and the independent set problem in RDV graphs, but these involve different techniques and data structures and we refer the interested reader to [16].

The *perfect k-clique packing* problem is defined as follows. Given a graph G and an integer k, we want to test whether there exists a partition $V_1, V_2, ...V_{\frac{n}{k}}$ 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.

Dahlhaus and Karpsinski [10] studied perfect k-clique packing (which they called *perfect k-multidimensional matching*) and showed that if a strongly chordal graph has a perfect k-clique packing, then a natural greedy strategy (see Algorithm 3) finds it.

Algorithm 3 Greedy-algorithm for perfect k-clique packing

Input: A graph G with a vertex order $v_1, \ldots, v_n$
initialize the packing $M = \emptyset$ and mark all vertices as "unmatched"
for $i = 1, \ldots, n$ **do**
  if v_i *is "unmatched"* **then**
    if v_i *has at least* $k-1$ *unmatched neighbours* **then**
      let $v_{j_1}, v_{j_2}, \ldots, v_{j_{k-1}}$ be the $k-1$ smallest unmatched neighbours of v_i
      add $\{v_i, v_{j_1}, \ldots, v_{j_{k-1}}\}$ to M, and mark all these vertices as "matched"
    else return *"no perfect k-clique packing"*
return M

This algorithm will take $O(n+m)$ time when implemented in the obvious way. However, it again only requires to find, for each vertex v_i, the unmatched neighbours that come later in the order, and among those, the ones that minimize the index. We can perform this in an RDV graphs with our techniques by doing $k-1$ range minimum queries per vertex v_i. Since this marks k vertices as matched, we therefore do $O(n)$ range minimum queries in total, which means that the run-time in RDV graphs gets reduced to $O(n \log n)$. See Algorithm 4.

▶ **Theorem 10.** *Given an n-vertex graph G with an RDV representation with host-tree T, and given an integer k, we can test in $O(|T| + n \log n)$ time whether G has a perfect k-clique packing.*

Algorithm 4 Reformulated greedy-algorithm for perfect k-clique packing in RDV graphs

Input: An RDV graph G with a bottom-up enumeration vertex order $v_1, \ldots, v_n$
initialize the packing $M = \emptyset$ and mark all vertices as "unmatched"
initialize an empty priority search tree $\mathcal{P}$
$y_{curr} = \infty$
for $i = 1, \ldots, n$ **do**
  $y_{prev} \leftarrow y_{curr}$; $y_{curr} \leftarrow y(t(v_i))$
  if $(y_{curr} < y_{prev})$ **then**
    for *all vertices* v_q *with* $y_{prev} > y(b(y_q)) \geq y_{curr}$ **do**
      insert the tuple $[x(b(v_q)), q]$ of v_q into $\mathcal{P}$
  if v_i *is "unmatched"* **then**
    delete the tuple of v_i from $\mathcal{P}$
    for $\ell = 1$ *to* $k-1$ **do**
      perform a range minimum query in $\mathcal{P}$ w.r.t. the x-range of $\mathbf{h}(v_i)$
      if *this returns null* **then return** *"no perfect k-clique packing"*
      else let $[\cdot, i_\ell]$ be the returned tuple and delete it from $\mathcal{P}$
    add $\{v_i, v_{j_1}, \ldots v_{j_{k-1}}\}$ to M, and mark all these vertices as "matched"
return M

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. doi:10.1006/jagm.1996.0058.

2 C. Berge. *Graphs and Hypergraphs, 2nd edition*. North-Holland, 1976. Translated from *Graphes et Hypergraphes, Dunod, 1970*.

3 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. URL: https://cosc.brocku.ca/~rnishat/CCCG_2024_proceedings.pdf.

4 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.

5 G. Brodal, P. Davoodi, and S. Rao. Path minima queries in dynamic weighted trees. In *Algorithms and Data Structures, (WADS 2011)*, volume 6844 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2011. doi:10.1007/978-3-642-22300-6_25.

6 S. Chaplick. PQR-trees and undirected path graphs. Master's thesis, Department of Computer Science, University of Toronto, 2008. URL: https://hdl.handle.net/1807/118597.

7 S. Chaplick and J. Stacho. The vertex leafage of chordal graphs. *Discrete Applied Mathematics*, 168:14–25, 2014. doi:10.1016/j.dam.2012.12.006.

8 L. Chen, R. Kyng, Y. Liu, R. Peng, M. Gutenberg, and S. Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Commun. ACM*, 66(12):85–92, 2023. doi:10.1145/3610940.

9 P. Christiano, J. Kelner, A. Madry, D. Spielman, and S. Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 273–282, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993636.1993674.

10 E. Dahlhaus and M. Karpinski. Matching and multidimensional matching in chordal and strongly chordal graphs. *Discrete Applied Mathematics*, 84(1):79–91, 1998. doi:10.1016/S0166-218X(98)00006-7.

11 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. URL: https://api.semanticscholar.org/CorpusID:1289747.

12 M. Farber. *Applications of L.P. duality to problems involving independence and domination.* PhD thesis, Rutgers University, 1982.

13 M. Farber. Characterizations of strongly chordal graphs. *Discret. Math.*, 43(2-3):173–189, 1983. doi:10.1016/0012-365X(83)90154-1.

14 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. doi:10.1016/0095-8956(74)90094-X.

15 F. Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discret. Math.*, 13(3):237–249, 1975. doi:10.1016/0012-365X(75)90021-7.

16 P. Gokhale. Efficient algorithms for RDV graphs. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2025. URL: https://hdl.handle.net/10012/21770.

17 M. Golumbic, D. Rotem, and J. Urrutia. Comparability graphs and intersection graphs. *Discrete Mathematics*, 43(1):37–46, 1983.

18 M. C. Golumbic. *Algorithmic graph theory and perfect graphs.* Academic Press, New York, 1st edition, 1980.

19 J. Kelner, Y. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, page 217–226, USA, 2014. doi:doi.org/10.1137/1.9781611973402.16.

20 Y. Liang and C. Rhee. Finding a maximum matching in a circular-arc graph. *Information Processing Letters*, 45(4):185–190, 1993. doi:10.1016/0020-0190(93)90117-R.

21 M.-S. Lin and S.-H. Su. Counting maximal independent sets in directed path graphs. *Information Processing Letters*, 114(10):568–572, 2014. doi:10.1016/j.ipl.2014.05.003.

22 M.-S. Lin and C.-C. Ting. Computing the k-terminal reliability of directed path graphs. *Information Processing Letters*, 115(10):773–778, 2015. doi:10.1016/j.ipl.2015.05.005.

23 M.C. Lin and J. Szwarcfiter. Characterizations and recognition of circular-arc graphs and subclasses: A survey. *Discret. Math.*, 309(18):5618–5635, 2009. doi:10.1016/J.DISC.2008.04.003.

24 L. Lovász and M. D. Plummer. *Matching theory*, volume 29 of *Annals of Discrete Mathematics*. North-Holland Publishing Co., Amsterdam, 1986.

25 Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. doi:10.1137/0214021.

26 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. doi:10.1137/17M1120920.

27 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. doi:10.1109/SFCS.1980.12.

28 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.

29 C. Monma and V. Wei. Intersection graphs of paths in a tree. *J. Comb. Theory, Ser. B*, 41(2):141–181, 1986. doi:10.1016/0095-8956(86)90042-0.

30 C. Papadopoulos and S. Tzimas. Computing a minimum subset feedback vertex set on chordal graphs parameterized by leafage. *Algorithmica*, 86(3):874–906, 2024. doi:10.1007/S00453-023-01149-5.

31 C. Rhee and Y. Liang. Finding a maximum matching in a permutation graph. *Acta Informatica*, 32(8):779–792, 1995. doi:10.1007/BF01178659.

32 D. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 81–90, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007352.1007372.

33 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10:99–127, 1976. doi:10.1007/BF01683268.

34 V. Vazirani. A proof of the MV matching algorithm. *CoRR*, abs/2012.03582, 2020. URL: https://arxiv.org/abs/2012.03582.

A The graph of Figure 1

We claimed earlier that the graph in Figure 1 is neither a circular arc graph nor a permutation graph, and we briefly argue this here. It suffices to prove that this holds for an induced subgraph G of this graph, shown in Figure 5.

Figure 5 The graph G (an induced subgraph of the graph of Figure 1) and a circular arc representation of $G \setminus \{v_1\}$, with vertex-arc correspondences indicated by colors/dash-style.

Most of our argument considers only the graph $G \setminus \{v_1\}$. This is well-known not to be a comparability graph [18, Figure 5.1], and since permutation graphs are subgraphs of comparability graphs and closed under vertex-deletion, G is not a permutation graph.

Next observe that vertices $\{v_2, v_3, v_8\}$ form what is known as an *asteroidal triple*: any two of them can be connected via a path that avoids the neighbourhood of the third. No such structure can exist in an interval graph. In fact, $G \setminus \{v_1\}$ is known to be an obstruction for *Helly circular-arc graphs* [23], i.e., it does not have a circular arc representation where for every clique C the arcs of vertices in C all share a common point. Since $\{v_5, v_6, v_7\}$ is the only clique of size exceeding 2, therefore in any circular arc representation of $G \setminus \{v_1\}$ the three arcs of v_5, v_6, v_7 do not share a common point. To still have pairwise intersections, these three arcs together cover the entirety of the circle. But then we cannot add an arc for v_1 anywhere since it has no edge to any of these three vertices. Therefore G is not a circular arc graph.

B Proof of Theorem 4

We want to show that the greedy-algorithm, applied to a bottom-up enumeration of an RDV representation, gives a maximum matching. By a result of Dahlhaus and Karpinski [10], it suffices to show that the bottom-up enumeration is a *strong perfect elimination order*. We define this first.

▶ **Definition 11.** *For a vertex v, let $N[v]$ be the* closed neighbourhood of *v, i.e., the set consisting of v and all neighbours of v. A vertex order $v_1, \ldots, v_n$ is called a* strong perfect elimination order *if for any integers $i, j, k, \ell \in \{1, \ldots, n\}$ such that $i \leq j$, $i \leq k$, $k \leq \ell$, $v_k, v_\ell \in N[v_i]$ and $v_k \in N[v_j]$, we have $v_j \in N[v_\ell]$.*

Dahlhaus and Karpinski showed that the greedy-algorithm gives a maximum matching if performed on a strong perfect elimination order, so to prove Theorem 4, it suffices to show the following.

▶ **Lemma 12.** *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 $v_1, v_2, ..., v_n$ be a bottom-up enumeration of G, and fix i, j, k, ℓ as in Definition 11. First, observe that (since all paths are downward) if $v_a \in N[v_b]$ for some $a \leq b$, then $t(v_a) \in P(v_b)$. Therefore $t(v_i) \in P(v_k)$, $t(v_i) \in P(v_\ell)$, so $P(v_k) \cap P(v_\ell) \supseteq \{t(v_i)\}$ is non-empty, hence $v_k \in N[v_\ell]$ and $t(v_k) \in P(v_\ell)$ by $k \leq \ell$. We have two cases:
Case 1: $k \leq j$. This implies that $t(v_k) \in P(v_j)$. As we already know that $t(v_k) \in P(v_\ell)$, this makes $P(v_j) \cap P(v_\ell) \supseteq \{t(v_k)\}$ non-empty and $v_\ell \in N[v_j]$ as required.
Case 2: $j < k$. This implies that $t(v_j) \in P(v_k)$ is a descendant of $t(v_k) \in P(v_\ell)$. By $i \leq j$ also $t(v_i) \in P(v_j)$ is a descendant of $t(v_j)$. Going upward in the tree from $t(v_i)$, we hence begin at a node in $P(v_\ell)$, go upward to $t(v_j)$, and further upward to $t(v_k)$ which is in $P(v_\ell)$. Since $P(v_\ell)$ is connected, therefore $t(v_j) \in P(v_\ell)$ and $v_j \in N[v_\ell]$ as desired. ◀