The Remove-r Distance-d Independent Set Problem

ICA-6077560
Ruben Meerkerk

Utrecht University

Utrecht University
Department of Information and Computing Sciences
Computing Science

May 16, 2019

Abstract

In this paper, we study the Maximum Remove-r Distance-d Independence Set (MRrDdIS for short) problem. The goal of this problem is to maximize the size of a distance d-independent set by removing at most a specified number r of vertices from a given graph. This problem generalizes the Maximum Distance-d Independent Set problem, where we look for a maximum size set of vertices for which the minimum distance between every pair of vertices from the set is at least d. The latter problem in turn generalizes the well known Maximum Independent Set problem. From this problem we derived two more problems, which both revolve around finding sets of nodes that we want to remove. One of them, called the d-Minimum Removal Set (d-MRS for short) problem, is about finding a minimum set of nodes to remove such that a given Independent Set becomes a Distance-d Independent Set on a given graph. The other one, called the (r, d)-Optimal Removal Set ((r, d)-ORS for short) problem, is about finding a set of at most r nodes that when removed from a graph maximizes its Maximum Distance-d Independent Set. In this paper, we prove that the decision variants of the MRrDdIS problem and the d-MRS problem are NP-complete. We present an algorithm for the MRrDdIS problem that in the worst case runs in $O(r^2n^3)$ time on trees. For the d-MRS problem we not only present a linear time algorithm, but also prove that a linear time algorithm can be generated for graphs with bounded treewidth.

1 Introduction

The Independent Set (IS) problem is a very well studied problem and one of the first to be proven NP-complete. The Distance-d Independent Set (DdIS) problem is a generalization of the Independent Set problem. A set is called a DdIS if each pair of elements is at least distance d apart. If $d = 2$, then it becomes the IS problem. Unlike the IS problem, the DdIS problem is not as well know. Most of the research done on the DdIS problem is done in the last two decades. Hence, the number of papers on this topic is not very large.

The problem that is considered in this thesis is the Remove-r Distance-d Independent Set (RrDdIS) problem. The maximization variant is called the Maximum Remove-r Distance-d Independent Set (MRrDdIS) problem. The RrDdIS problem is a generalization of the DdIS problem, in which we allow the removal of r nodes. By removing nodes from the graph, we try to get a new graph for which we can find an even larger DdIS. If $r = 0$ then it becomes the DdIS problem again, but if $r = n$ (n being the amount of nodes in the graph) then it becomes the IS problem. The latter is due to the fact that for any IS we can remove its neighbourhood to completely isolate every node in the set.

Finding the right set of nodes to remove can be seen as a whole new problem on its own. The (r, d)-Removal Set ((r, d)-RS) problem is about checking if a given set is an RrDdIS by finding the right set of nodes to remove, also known as the (r, d)-RS. The minimization variant of this problem is the d-Minimum Removal Set (d-MRS) problem. For this problem we want to find the smallest set of nodes to remove from a graph to turn some given set into a DdIS. The (r, d)-Optimal Removal Set ((r, d)-ORS) problem is about finding a set of at most r nodes, that when removed creates the graph with the largest Maximum Distance-d Independent Set (MDdIS). In other words, we want to find the (r, d)-RS of some MRrDdIS. To the best of our knowledge, neither the RrDdIS problem nor any of these removal set problems have been studied before.

1.1 Previous work

While there seems to be no previous work done on the MRrDdIS problem, there are several papers written about the MDdIS problem, most of which very recent. The MDdIS problem was first introduced introduced by Eto et al. [9]. They proved that the decision variant was NP-complete even for planar bipartite graphs. Before they introduced the problem, Bhattacharya

and Houle [4] already solved the MDdIS problem for trees. Since their main focus was solving something called the dispersion problem, we do not consider their work as an introduction to this problem. The dispersion problem is about finding a set of size at least some given $k \geq 0$ such that the distance of the closest vertex pair in the set is maximized. Work on this problem has also been done by [8, 16, 18].

Agnarsson et al. [1] observed that solving the MDdIS problem on a graph $G = (V, E)$ is equivalent to solving the MIS problem on the power graph $G^{d-1} = (V, E^{d-1})$ such that E^{d-1} contains an edge between every vertex pair that has a shortest path of distance at most $d-1$. They proved that it is possible to efficiently compute the power graph of certain geometric intersection graphs in such a way that the power graph still belongs to the same class of geometric intersection graph. Since polynomial time algorithms solving the MIS problem already existed for these kind of graphs, it follows that a polynomial time algorithm for the MDdIS problem is possible for these graphs as well.

Unfortunately there are many graph classes for which powers of instances of the class not necessarily belong to the class. For example the power graph of a bipartite graph with a connected component with at least two edges is never bipartite. Even worse, as proven by Eto et al. [9] for $d \geq 3$, the DdIS problem is NP-complete, even though the MIS problem was shown to be polynomially solvable on bipartite graphs by Harary [12]. NP-completeness was even proven for planar bipartite graphs with a maximum degree of 3. Chordal graphs on the other hand remain chordal graphs for odd powers. Since the MIS problem is polynomial solvable on chordal graphs, we can polynomial solve the MDdIS problem for any fixed even integer $d \geq 2$. It was also proven that for all fixed odd integers $d \geq 3$ the problem would be NP-complete.

The power graph of a tree is not a tree. Fortunately, as shown by Bhattacharya and Houle [4], the MDdIS problem can still be solved for trees in subquadratic time. It was also shown by Turau and Köhler [19] that for any odd integer $d \geq 3$ we can also solve the MDdIS problem in polynomial time using a distributed algorithm.

For several graph parameters, there exist polynomial time algorithms for several problems on graph classes where the parameter is bounded by some constant, see e.g. [13, 15, 17]. Also, work has been done by Bacso et al. [2, 3] on subexponential algorithms for H-free graphs. Approximation schemes for

certain graph classes were found by [10, 11, 14] or proven that certain approximation schemes were not possible unless P = NP.

1.2 Outline

In Section 2 we will give definitions for the different generalizations of the Independent Set problem and the related Removal Set problems. After that we look in Section 3 at the complexity of some of those problems for general graphs. In Section 4 we prove using Courcelle's theorem that the d-MRS problem can be solved in linear time on graphs with bounded treewidth. In Section 5 we give an algorithm that solves the d-MRS problem on trees (which have bounded treewidth) in linear time, but also guarantee the algorithm to run logarithmically in constant d. In Section 6 an algorithm is described that solves the MRrDdIS problem for trees in polynomial time. Section 7 will summarize the found results, but also list directions for future research.

2 Definitions and preliminaries

Let $G = (V, E)$ be a graph for which V is the set of vertices and E the set of edges. We will assume throughout the paper that $|V| = n$ and $|E| = m$. We also denote the set of vertices of any graph G with $V(G)$.

Let $\text{dist}_G(u, v)$ denote the shortest path distance between vertices u and v (the minimum number of edges of a uv-path) in G.
Let $S \subseteq V$, then $\text{dist}_G(S, v) = \text{dist}_G(v, S) = \min_{u \in S}(\text{dist}_G(u, v))$.
Let also $S' \subseteq V$, then $\text{dist}_G(S, S') = \min_{u \in S, v \in S'}(\text{dist}_G(u, v))$.
The diameter of G, denoted with $\text{diam}(G)$, is the longest shortest path distance in G. More formally: $\text{diam}(G) = \max_{u,v \in V(G)}(\text{dist}_G(u, v))$.

Let $S \subseteq V$, then $G[S] = (S, E \cap (S \times S))$ is the subgraph of G induced by S.
A component C is a connected subgraph of some graph G.

$N(v) = \{v' \in V(G) \mid \text{dist}_G(v, v') = 1\}$ is the open neighbourhood of a vertex $v \in V(G)$.
$N[v] = \{v\} \cup N(v)$ is the closed neighbourhood of a vertex $v \in V(G)$.
$N(S) = \{v' \in V(G) \setminus S \mid \exists v \in S : \text{dist}_G(v, v') = 1\}$ is the open neighbourhood of a subset $S \subseteq V(G)$.
$N[S] = S \cup N(S)$ is the closed neighbourhood of a subset $S \subseteq V(G)$.

2.1 Graph classes

A tree $T = (V, E)$ is an undirected graph that is both connected and acyclic. A rooted tree $T = (V, E)$ is a tree with a single root node/vertex denoted with $\text{root}(T) \in V$. In most cases, when working with rooted trees, we drop the word "rooted" for brevity. A node/vertex $v \in V$ of a rooted tree is an ancestor of some node/vertex $u \in V$, if $\text{dist}_T(u, \text{root}(T)) = \text{dist}_T(u, v) + \text{dist}_T(v, \text{root}(T))$. If v is an ancestor of u, then u is a descendant of v. This means that a node v is both its own ancestor and descendant. If u and v are also adjacent, then v is the parent of u and u is a child of v. Each node/vertex v in the tree has a set of children denoted with $C_T(v) = \{v_1, v_2, v_3, ..., v_k\}$ such that k is the amount of children of v. We will often use the notation $C(v)$ instead, because it will be clear from the context which tree T is meant. If $|C(v)| = 0$, then v is a leaf. If $|C(v)| = 1$, then v is a linkpoint. If $|C(v)| \geq 2$, then v is a branchpoint. A subtree of $T = (V, E)$ can be denoted as $\text{tree}_T(v) = (V', E')$ such that V' contains all descendants of v in T, $E' = E \cap (V' \times V')$, and $v = \text{root}(T')$. The height of a tree is zero if it only consists of a single node, else the height is: $\text{height}(T) = 1 + \max(\text{height}(T') | T' = \text{tree}_T(v') \; \forall v' \in C(v))$.

A graph $G = (V, E)$ is p-partite if $V = \bigcup_{i=1}^{p} V_i$ and $E \subseteq \{V_i \times V_j | \forall 1 \leq i < j \leq p\}$ such that $\forall 1 \leq i \leq p$ we have that $V_i \neq \emptyset$.
If $p = 2$, then the graph is called bipartite and can also be denoted as $B = (V_1, V_2, E)$.
If $p = 3$, then the graph is called tripartite and can also be denoted as $G = (V_1, V_2, V_3, E)$.

2.2 Problem definitions

Let $G = (V, E)$ be a given graph, then:

- A set $S \subseteq V$ is a DdIS (Distance-d Independent Set) of G, if for any pair $u, v \in S$ the distance $\text{dist}_G(u, v) \geq d$.
- A set $M \subseteq V$ is a MDdIS (Maximum Distance-d Independent Set) of G, if M is a DdIS of G and for any DdIS S of G we have that $|M| \geq |S|$.
- A set $S \subseteq V$ is a IS (Independent Set) of G, if S is a D2IS.
- A set $M \subseteq V$ is a MIS (Maximum Independent Set) G, if M is a MD2IS.

- A set $S \subseteq V$ is a RrDdIS (Remove-r Distance-d Independent Set) of G, if there exists a set $R \subseteq V \setminus S$ such that $|R| \leq r$ and S is a DdIS of $G[V \setminus R]$.
- A set $M \subseteq V$ is a MRrDdIS (Maximum Remove-r Distance-d Independent Set) of G, if M is a RrDdIS of G and for any RrDdIS S of G we have that $|M| \geq |S|$.
- A set $R \subseteq V$ is an (r, d)-RS ((r, d)-Removal Set) of a subset $S \subseteq V$, if $R \cap S = \emptyset$, $|R| \leq r$, and S is a DdIS of $G[V \setminus R]$.
- A set $M \subseteq V$ is a d-MRS (d-Minimum Removal Set) of a subset $S \subseteq V$, if M is a $(|M|, d)$-RS of S and for any $r < |M|$ there exists no (r, d)-RS for S.
- A set $R \subseteq V$ is an (r, d)-ORS ((r, d)-Optimal Removal Set) of G, if it is an (r, d)-RS of some MRrDdIS of G.

Let $T = (V, E)$ be a given tree, then:

- A set $R \subseteq V$ is a d-SMRS (Special Minimum Removal Set) of a subset $S \subseteq V$, if R is a d-MRS of S and if a d-MRS M exists for S such that if $\text{root}(T) \in M$, then $\text{root}(T) \in R$, else we have for any d-MRS M that $d_{T[V \setminus R]}(\text{root}(T), S) \geq d_{T[V \setminus M]}(\text{root}(T), S)$.
- A set $S \subseteq V$ is a (v, i, j, d)-Cell of T, if S is an RjDdIS of $\text{tree}_T(v)$ and there is a (j, d)-RS R of S such that either $v \in R$ or $\forall s \in S$ we have that $d_{T[V \setminus R]}(v, s) \geq i$.
- A set $M \subseteq V$ is a (v, i, j, d)-MaxCell of T, if M is a (v, i, j, d)-Cell of T and for any (v, i, j, d)-Cell S of T we have that $|M| \geq |S|$.
- A set $A_i = \{A_{i\ 0}, A_{i\ 1}, ..., A_{i\ r}\}$ is a (v, i, r, d)-Row of T, if $\forall\ 0 \leq j \leq r$ $A_{i\ j}$ is a (v, i, j, d)-Cell of T.
- A set $M_i = \{M_{i\ 0}, M_{i\ 1}, ..., M_{i\ r}\}$ is a (v, i, r, d)-MaxRow of T, if $\forall\ 0 \leq j \leq r$ $M_{i\ j}$ is a (v, i, j, d)-MaxCell of T.
- A set $A = \{A_0, A_1, ..., A_{d-1}\}$ is a (v, r, d)-Array of T, if $\forall\ 0 \leq i \leq d-1$ A_i is a (v, i, r, d)-Row of T.
- A set $M = \{M_0, M_1, ..., M_{d-1}\}$ is a (v, r, d)-MaxArray of T, if $\forall\ 0 \leq i \leq d-1$ M_i is a (v, i, r, d)-MaxRow of T.

- A set $A = \{A_1, A_2, ..., A_k\}$ is a (v, p, r, d)-MaxRowSet of T, if $1 \leq k \leq |C(v)|$, $\forall\ 1 \leq i \leq k$ we have that A_i is a (v_i, p_i, r, d)-MaxRow of T, $v_i \in C(v)$, $p_i + 1 \geq p$, and that $\forall i \neq j \in [1, k]$ we have that $p_i + p_j + 2 \geq d$.
- A set $M = \{M_0, M_1, ..., M_r\}$ is a SemiMaxRow of A, if A is a (v, i, r, d)-MaxRowSet of T and $\forall 0 \leq j \leq r$ we have that $|M_j| \geq |S|$ for every $S = \bigcup_{k=1}^{|A|} A_{k\ q_k}$ such that $0 \leq \sum_{k=1}^{|A|} q_k \leq j$.

The problems below are just further generalizations of the DdIS problems as defined in [9]. It is probably worth noting that r is one of the input variables, meaning that r is not a constant. If r were a constant parameter like d, then it might be possible to solve the RrDdIS problem by just trying to remove $O(c^r)$ possible combinations of r vertices with constant $c > 1$ and then solving the DdIS or MDdIS problem on the remaining graph. This might be interesting to look into as well, but it would also mean that researching the RrDdIS problem would be redundant, considering that we can simply research the DdIS problem instead. Of course this is based on the conjecture that we only need to look for $O(c^r)$ different sets to remove.

RrDdIS PROBLEM	
Input:	A graph $G = (V, E)$ and positive integers k and r.
Question:	Does G contain a RrDdIS of size k or larger?

MRrDdIS PROBLEM	
Input:	A graph $G = (V, E)$ and a positive integer r.
Output:	A MRrDdIS of G.

The problems below are somewhat different from the ones above. These problems are derived from the observation that one could divide the RrDdIS problems into a part where we find a set of vertices that we want to remove from the graph and a part were we solve the DdIS problem on the new graph. Even though they are derived from the above problems, they are not quite the same. The (r, d)-RS problem is a decision problem. It decides whether a given set S is a RrDdIS, by checking if an (r, d)-RS exists for S. The d-MRS problem is a minimization version of this problem. Here we look for the smallest (r, d)-RS for S, which is called a d-MRS. The (r, d)-ORS problem is an optimization problem. It does not take a set S as input like the other two problems. The goal of the algorithm is to find an (r, d)-RS for some

MRrDdIS, which is called an (r, d)-ORS. Note that this problem does not try to optimize the size of the output set.

(r, d)-RS PROBLEM	
Input:	A graph $G = (V, E)$, a subset $S \subseteq V$, and a positive integer r.
Question:	Does G contain an (r, d)-RS for S.

d-MRS PROBLEM	
Input:	A graph $G = (V, E)$ and an Independent Set $S \subseteq V$.
Output:	A d-MRS of S.

(r, d)-ORS PROBLEM	
Input:	A graph $G = (V, E)$ and a positive integer r.
Output:	An (r, d)-ORS of G.

3 Complexity of the Problem

The IS problem and its generalization the DdIS problem are known to be NP-complete for general graphs. The RrDdIS problem is clearly an NP-complete problem as well since it is a generalization of the DdIS problem, but in Theorem 1 we will see that it is also true when $r \neq 0$.

Theorem 1. *The RrDdIS problem is NP-complete for any $d \geq 2$ and $r \geq 0$ for general graphs.*

Proof. If $d = 2$, then the RrDdIS problem becomes the RrD2IS problem. It is clear that a RrD2IS is an IS and vice versa, which means that the RrD2IS problem is the same as the IS problem, which is a known NP-complete problem.
We can prove NP-completeness for $d \geq 3$ by a reduction from the DdIS problem. An instance of the DdIS problem will have as input a graph $G = (V, E)$ and a positive integer k. We transform the graph instance into $G' = (V', E')$ such that $V' = V \cup C$, $E' = E \cup (V' \times C)$, and C is a set of vertices such that $C \cap V = \emptyset$ and $|C| = r$.
We claim that any RrDdIS of size greater or equal to 2 of G' is a DdIS of G and vice versa and prove it as follows. Let $G'' = G'[V \setminus R]$ such that

R is some subset of V and $|R| \leq r$. If $R \neq C$, then $\text{diam}(G'') \leq 2$. This would mean that any DdIS of G'' would have a maximum size of 1. So if a RrDdIS of G' has size greater or equal to 2, then it can only be a DdIS of G'' if $R = C$. We will therefore assume from now on that $R = C$, which means that $G'' = G$. By definition we know that any DdIS of $G'' = G$ must also be a RrDdIS of G'. We also know that any RrDdIS of G' of size greater or equal to 2 must be a DdIS of $G'' = G$. So we have proven our claim.
Since any RrDdIS of size greater or equal to 2 of G' is also a DdIS of G and vice versa, we know that for $k \geq 2$ that (G', k) is a YES-instance of the RrDdIS problem if and only if (G, k) is a YES-instance of the DdIS problem. □

According to Theorem 2 the (r, d)-RS problem is NP-complete too when $d \geq 4$ and the Independent Set S has size greater than 2. The proof relies on the the Vertex Cover problem being NP-complete on tripartite graphs, which we proved in Lemma 3. When $d = 2$ it is trivial to solve, because the empty set will always be the 2-MRS. When $d = 3$ the problem is trivial as well, because we only have to list all nodes that neighbour at least two elements of S. If the resulting set is larger than r, then we know that no $(r, 3)$-RS exists for S. According to Lemma 2, we can solve the 4-MRS problem in polynomial time if $|S| = 2$. This relies on the well known fact that the Vertex Cover problem is polynomial solvable for bipartite graphs. The (r, d)-RS problem is probably NP-complete for $d > 4$ and $|S| = 2$, but a proof has not been found yet.

Lemma 1. *C is a Vertex Cover of p-partite graph $G = (V, E)$ $\iff$ M is a $(|C|, d) - RS$ of S for graph $G' = \left(S \cup V, E \cup \left(\bigcup_{i=1}^{p} \{s_i\} \times V_i\right)\right)$ with $S \cap V = \emptyset$, $d \geq 4$, and such that $\forall 1 \leq i < j \leq p$ we have that $s_i \neq s_j \in S$.*

Proof. If C is a Vertex Cover of G, then that means that by definition $\forall (u, w) \in E$ that $u \in C \vee w \in C$. This means that $G'[S \cup V \setminus C] = \left(S \cup V \setminus C, \bigcup_{i=1}^{p} \{s_i\} \times V_i\right)$. This means that after removing C from G' that no paths are left from any element of S to any other element of S. This makes C a $(|C|, d)$-RS of S for G'.
Suppose C is not a Vertex Cover of G, then there must be an edge $(u, w) \in E$ such that $u \notin C \wedge w \notin C$. Let i and j be such that $u \in V_i$ and $w \in V_j$. We know that $i \neq j$, because $E \cap V_i \times V_i = \emptyset$. This means that $< s_i, u, w, s_j >$ is a path of length 3. Since C does not contain any of those nodes, removing C from G' will mean that a path of length 3 between two elements of S remains. This means that C is not a $(|C|, d)$-MRS of S for G'. □

Lemma 2. *For a given graph $G = (V, E)$ and Independent Set $S \subseteq V$ such that $|S| = 2$, we can find the 4-MRS of S in polynomial time.*

Proof. Let $S = \{s_1, s_2\}$ and $R = N(s_1) \cap N(s_2)$. All elements of R have to be elements of the 4-MRS. Let $B = (V_1, V_2, F)$ be a bipartite graph such that $V_1 = N_G(s_1) \setminus R$, $V_2 = N_G(s_2) \setminus R$, and $F = E \cap (V_1 \times V_2)$. $G[V \setminus R]$ is a graph for which all paths between s_1 and s_2 are of length at least 3. Since we are only interested in the removal of nodes of paths with length at most 3, we can ignore all edges and nodes that are not on those paths. Graph $G' = (S \cup V_1 \cup V_2, F \cup (\{s_1\} \times V_1) \cup (\{s_2\} \times V_2))$ only contains the nodes and edges that are part of some path between s_1 and s_2 of length 3. This means that a set M is a 4-MRS of G' if and only if M is a 4-MRS of $G[V \setminus R]$. We know from Lemma 1 that a bipartite graph B can constructed for G' such that any MVC on the bipartite graph is also a 4-MRS. □

Lemma 3. *The Vertex Cover problem for tripartite graphs is NP-complete.*

Proof. We will prove this through a reduction from the Vertex Cover problem for general graphs. Let graph $G = (V, E)$ and positive integer $k \geq 0$ together be an input instance of the Vertex Cover problem. Let $G' = (V', E')$ such that $V' = V \cup V_1 \cup V_2$, $V_1 = \{v_1^e | \forall e \in E\}$, $V_2 = \{v_2^e | \forall e \in E\}$, and $E' = \{(u, v_1^e), (v_1^e, v_2^e), (v_2^e, w), | \forall (u, w) = e \in E\}$. Clearly G' is a tripartite graph where V' partitions into V, V_1, and V_2. Let tripartite graph G' and positive integer $k' = k + |E|$ together be an input instance of the Vertex Cover problem for tripartite graphs. We claim that a Vertex Cover C exists for G such that $|C| \leq k$ if and only if a Vertex Cover C' exists for G' such that $|C'| \leq k + |E|$. Assume that C is a Vertex Cover of G such that $|C| \leq k$. Then let $C' = C \cup C_1 \cup C_2$ such that $C_1 = \{v_1^e \in V_1 | \forall (u, w) = e \in E \wedge w \in C\}$ and $C_2 = \{v_2^e \in V_2 | \forall (u, w) = e \in E \wedge v \in C \wedge w \notin C\}$. We know that $|C_1| + |C_2| = |E|$, since C covers all edges of E. This means that $|C'| = |C| + |C_1| + |C_2| \leq k + |E|$. We also know that C' is a vertex cover, because $\forall (u, w) = e \in E$ we get that:

- If $w \in C$, then edges (u, v_1^e) and (v_1^e, v_2^e) are covered by v_1^e and edge (v_2^e, w) is covered by w. Note that (u, v_1^e) could be covered by u as well.
- If $u \in C \wedge w \notin C$, then edge (u, v_1^e) is covered by u and edges (v_1^e, v_2^e) and (v_2^e, w) are covered by v_1^e.

Assume that C' is a Vertex Cover of G' such that $|C'| \leq k + |E|$. Let $C = V \cap C''$ such that $C'' = (C' \setminus V_1') \cup U$, $V_1' = \{v_1^e \in V_1 | \forall (u, w) = e \in E \wedge u, w \notin C'\}$, and $U = \{u \in V | \forall (u, w) \in E \wedge u, w \notin C'\}$. We know that whenever $(u, w) = e \in E \wedge u, w \notin C'$ that $v_1^e, v_2^e \in C'$, because C' is a Vertex Cover of G' and has to cover edges (u, v_1^e) and (v_2^e, w). This means that

$|C''| = |C'|$ and $|(V_1 \cup V_2) \cap C''| \geq |E|$. Clearly C is a vertex cover of G. We also know that $|C| = |V \cap C''| = |C''| - |(V_1 \cup V_2) \cap C''| \leq |C'| - |E| \leq k$. □

Theorem 2. *The (r, d)-RS problem with $d \geq 4$ and $|S| \geq 3$ is NP-complete for general graphs.*

Proof. We will prove this through a reduction from the Vertex Cover problem for tripartite graphs. We know from Lemma 3 that the Vertex Cover problem is NP-complete for tripartite graphs. Let tripartite graph $G = (V_1, V_2, V_3, E)$ and integer $k \geq 0$ together be the input instance of the Vertex Cover problem. Let graph $G' = (S \cup V_1 \cup V_2 \cup V_3, E \cup (s_1 \times V_1) \cup (s_2 \times V_2) \cup (s_3 \times V_3))$ and $s_1, s_2, s_3 \in S$ such that $s_1 \neq s_2 \neq s_3 \neq s_1$ together be the input instance of the (k, d)-RS problem. From Lemma 1 we can conclude that we can find a (k, d)-RS of S for G' if and only if there is a Vertex Cover of size at most k in G. □

4 The d-MRS Problem on Graphs with Bounded Treewidth

Theorem 1 proves that the (r, d)-RS problem is NP-complete in the general case, making the d-MRS problem NP-hard. It is still possible though to solve the d-MRS problem in linear time for certain graphs. Courcelle [7] states that every graph property that can be defined in MSOL (monadic second-order logic) can be decided in linear time on graphs of bounded treewidth. This also known as Courcelle's theorem. Borie et al. [5] present similar results, but independent from Courcelle [7]. A survey was also made by Borie et al. [6] on the topic, in which they show how these results can be used for various well known NP-complete problems and graph classes.

We base our result on the terminology and description of the results by Borie et al. [5]. For details, including the definition of what a regular graph property is, we refer to [5]. We start by giving a number of results from [5] upon which we build.

Theorem 3 (Borie et al. [5, Theorem 1])**.** *Each of the following predicates is regular:*

1. *$v_1 = v_2$ (vertex equality).*
2. *$Inc(v_1, e_1)$ (vertex-edge incidence).*
3. *$v_1 \in V_1$ (vertex membership).*

4. $e_1 \in E_1$ *(edge membership).*

Theorem 4 (Borie et al. [5, Theorem 2])**.** *The set of regular predicates is closed under* $\neg$, $\wedge$, $\vee$, $\forall$, *and* $\exists$, *where quantification is over variables which range over vertices, edges, vertex sets, and edge sets.*

Theorem 5 (Borie et al. [5, Theorem 3])**.** *If* P *and* Q *are regular properties, then each of the following predicates is a regular property:*

$$\begin{aligned} P \to Q &\iff \neg P \wedge Q. \\ P \leftrightarrow Q &\iff (P \to Q) \wedge (Q \to P). \\ (\exists x \in X)(P(x)) &\iff (\exists x)(x \in X \wedge P(x)). \\ (\forall x \in X)(P(x)) &\iff (\forall x)(x \in X \to P(x)). \\ Q(P(x)) &\iff (\exists y)((y = P(x)) \wedge Q(y)). \end{aligned}$$

Theorem 6 (Borie et al. [5, Theorem 4])**.** *Each of the following predicates is a regular property:*

1. $V_1 \setminus V_2 = V_3$.
2. $V_1 \subseteq V_2$.
3. $|V_1| \geq m$.
4. $Path(V_1, E_1)$ *(this property expresses that the vertices in* V_1 *and the edges in* E_1 *form a path together.).*

Theorem 7 (Borie et al. [5, Theorem 5])**.** *If* x *denotes a set and* $P(x)$ *is regular, then the following problem can be solved in linear time on graphs with bounded treewidth:*

$$\min |x_1| : P(x_1).$$

Theorem 8. *A linear time algorithm can be constructed to solve the d-MRS problem on graphs with bounded treewidth.*

Proof. Let us define the d-MRS problem for given graph $G = (V, E)$ and IS $S \subseteq V$ as follows:

$$\min |R| : d\text{-RS}(R, S, V, E).$$

Using Theorems 3, 4, 5, and 6, we can show that d-RS(R, S, V, E) is a regular property as follows:

$$\begin{aligned}
E' = E - V &\iff (E' \subseteq E) \wedge (\forall e \in E) \\
&\qquad (e \in E' \leftrightarrow (\forall v \in V)(\neg \mathrm{Inc}(v, e))). \\
\mathrm{dist}(v_1, v_2, V, E) \geq d &\iff (\forall V' \subseteq V)(\forall E' \subseteq E) \\
&\qquad (((v_1, v_2 \in V') \wedge \mathrm{Path}(V', E')) \rightarrow |V'| \geq d). \\
\mathrm{D}d\mathrm{IS}(S, V, E) &\iff (\forall s_1 \neq s_2 \in S)(\mathrm{dist}(s_1, s_2, V, E) \geq d). \\
d\text{-RS}(R, S, V, E) &\iff (R \subseteq (V \setminus S)) \wedge (V' = V \setminus R) \wedge (E' = E - R) \\
&\qquad \wedge \mathrm{D}d\mathrm{IS}(S, V', E').
\end{aligned}$$

Each of the above predicates on the left is proven to be a regular property by the MSOL on the right. Note that predicate d-RS(R, S, V, E) is true if and only if R is a $(|R|, d)$-RS of S on $G = (V, E)$. This means that the minimization problem we defined is equivalent to the actual d-MRS problem. So according to Theorem 7, we can solve the d-MRS problem in linear time on graphs with bounded treewidth. □

In the next section we will show how to solve the d-MRS problem in linear time for trees. Trees have bounded treewidth, which means that we already know how to construct a linear time algorithm for trees automatically. Note that the algorithms constructed by Courcelle's theorem are only linear in the graph size. These algorithms will most likely have a factor that is exponential in d for trees or other graphs with bounded treewidth. In Section 5 we will present an algorithm that has a factor sub-linear in d.

5 The d-MRS Problem on Trees

Solving the d-MRS problem for a given tree $T = (V, E)$ and set $S \subseteq V$ can be done in linear time, as shown in Section 4. The algorithm we present in this section will also guarantee that it runs with a factor that is logarithmic in d. The algorithm will achieve this by bottom-up computing values, ranging from 0 to d, for all the nodes in a given tree. The values represent more or less the distance to the closest descendant that is part of S. If a node v itself is an element of S, then v is itself the closest descendant that is part of S and gets therefore a value of 0.

For a node $v \in V$, we do not need to know exactly how far the closest descendant u is, if u is really far away. If $d_T(u, v) \geq d - 1$, then we know that for any node $w \notin \mathrm{tree}_T(v)$ that $d_T(u, w) \geq d$. In other words, w does not

need to be separated from u. This means that node u can be ignored when assigning values to nodes outside $\text{tree}_T(v)$. So instead of assigning $d_T(u, v)$ to v, we will assign $d - 1$ to v. For example, a leaf or any node $v \in V$ that has no descendant $u \in S$ would normally be assigned ∞, but now it will be assigned $d - 1$.

If $v \in V$ is a branchpoint and there are at least two descendant nodes $u, w \in S$ such that $d(u, v) + d(v, w) < d$, then those nodes are too close to each other and have to be separated. By removing v we separate u and w and may also separate other descendant nodes in S from each other and from ancestor nodes in S. Instead of actually removing v, we will assign it a value of d. When a node v is assigned a value of d, we know that it will be removed, because otherwise it would have been given a value between 0 and $d-1$. Because we are removing v, any descendant of v can be ignored when assigning values to nodes outside $\text{tree}_T(v)$.

Note that the algorithm will not only choose elements with value d to remove. If v has a value less than $d - 1$ assigned to it and its parent is an element of S, then the algorithm will remove that element as well.

To calculate the values for each node as intended, we will use a function $q : V \to [0, d]$. Let v be any node in the tree, $C(v) = \{v_1, v_2, v_3, ..., v_k\}$ such that k is the amount of children of v and $q(v_i) \leq q(v_{i+1})$ for all $0 < i \leq k$, then we compute $q(v)$ as follows:

$$q(v) := \begin{cases} 0, & \text{if } v \in S, \\ d - 1, & \text{if } C(v) = \emptyset, \\ min(q(v_1) + 1, d - 1), & \text{if } C(v) = \{v_1\}, \\ d, & \text{if } q(v_1) + q(v_2) + 2 < d, \\ min(q(v_1) + 1, d - 1), & \text{otherwise.} \end{cases} \tag{1}$$

After calculating all the values bottom up in $O(n)$ time, we will add certain vertices to a set R. All vertices with a value of d will be added to R. Vertices with a value less than $d - 1$ will be added to R, if their parent has a value of 0. After this is done, the algorithm will return R as the d-MRS of S, assuming that S is an independent set.

Algorithm 1: TMRS(T, S, d)

Input: A tree T, IS $S \subseteq V$ and positive integers r and d.
Result: A d-MRS of S.
$R \leftarrow \emptyset$;
compute $q(v)$ bottom up for all $v \in V$;
forall $v \in V$ **do**
  if $q(v) = d$ **then**
    $R \leftarrow R \cup \{v\}$;
  end
  else if $q(v) < d - 1$ *and* v *has a parent* w *such that* $q(w) = 0$ **then**
    $R \leftarrow R \cup \{v\}$;
  end
end
return R;

In Figure 1 we see an example of what the algorithm will do for a given tree and IS S. The result is a 5-MRS $R = \{v_3, v_5, v_{15}\}$ of size 3. It is easy to see that a smaller set is not possible in this example. Node v_5 is the only node connecting v_{10} and v_{11}, so it must be removed to separate those two nodes. The same can be said of v_{15}, which is the only node connecting v_{10} and v_{18}. Node v_3 is also necessary, because it is the only node connecting v_7 with both v_{12} and v_{17}.

A node $u \in V$ is a (d, S)-descendant of v, if u is a descendant of v, $u \in S$, $d(v, u) < d - 1$, and $\forall w \in V : d(u, v) = d(u, w) + d(w, v)$ we have that $q(w) \neq d$.

Lemma 4. *If $q(v) < d-1$, then there is a node u that is the (d, S)-descendant of v, closest to v, such that $d(u, v) = q(v)$. If $q(v) \geq d - 1$, then v has no (d, S)-descendant.*

Proof. We can prove the above lemma through induction.

1. If $v \in S$, then $q(v) = 0 < d - 1$. Since v is a (d, S)-descendant of itself, we have that $d(v, v) = 0 = q(v)$.

2. If $v \notin S$ and v is a leaf, then $q(v) = d - 1$. Since v is a leaf, v itself is its only descendant. But since $v \notin S$ it is not a (d, S)-descendant of itself. So v has no (d, S)-descendant.

3. If $v \notin S$ and v is a linkpoint with child w and we assume that above statement is true for w, then it follows that it is true for v as well for the following reasons:

   - If $q(w) < d-2$, then it follows that $q(v) = q(w)+1 < d-1$. Let u be the (d,S)-descendant of w, closest to w, then u is also the $(d-S)$-descendant of v, closest to v. We also know that $d(u,v) = d(u,w)+1 = q(w)+1 = q(v)$.
   - If $q(w) = d-2$, then it follows that $q(v) = q(w)+1 = d-1$. Let u be a (d,S)-descendant of w, then u is not a $(d-S)$-descendant of v, because $d(u,v) = d(u,w)+1 \geq d-1$. Since $v \notin S$ we know that v is not its own (d,S)-descendant, which means that v has no $(d-S)$-descendant.
   - If $q(w) \geq d-1$, then it follows that $q(v) = d-1$. Since w has no (d,S)-descendant and v is not its own (d,S)-descendant, we know that v has no (d,S)-descendant.

4. If $v \notin S$ and v is a branchpoint and we assume that above statement is true for all children of v, then it follows that it is true for v as well for the following reasons. Let v' be a child of v such that

$$q(v') \leq \min(q(v_i)|v_i \in C(v)).$$

   Let v'' be a child of v such that

$$q(v'') \leq \min(q(v_i)|v_i \in C(v) \setminus \{v'\}).$$

   - If $q(v')+q(v'')+2 < d$, then it follows that $q(v) = d$. Non of the descendants of v can be a (d,S)-descendant, because for all descendants u of v we have that $d(u,v) = d(u,v)+d(v,v)$ and $q(v) = d$.
   - If $q(v')+q(v'')+2 \geq d$ and $q(v') < d-2$, then it follows that $q(v) = q(v')+1 < d-1$. Let u be the (d,S)-descendant of v', closest to v', then u is also the $(d-S)$-descendant of v, closest to v. We also know that $d(u,v) = d(u,v')+1 = q(v')+1 = q(v)$.
   - If $q(v')+q(v'')+2 \geq d$ and $q(v') = d-2$, then it follows that $q(v) = q(v')+1 = d-1$. Let u be a (d,S)-descendant of any child w of v, then u is not a $(d-S)$-descendant of v, because $d(u,v) = d(u,w)+1 \geq d-1$. Since $v \notin S$ we know that v is not its own (d,S)-descendant, which means that v has no $(d-S)$-descendant.

- If $q(v') + q(v'') + 2 \geq d$ and $q(v') \geq d - 1$, then it follows that $q(v) = d - 1$. Since v' has no (d, S)-descendant, we know that non of the other children have one either. On top of that v is not its own (d, S)-descendant, so we know that v has no (d, S)-descendant.

5. From base cases 1 and 2, and induction steps 3 and 4 we can conclude that the above statement is indeed correct.

□

Lemma 5. *For a given tree $T = (V, E)$, IS $S \subseteq V$, and positive integer $d \geq 2$, if root$(T) \in S$, then any d-MRS of S is also a d-SMRS.*

Proof. Since root$(T) \in S$ we know that for any d-MRS R that root$(T) \notin R$ and $d_{T[V \setminus R]}(\text{root}(T), S) = 0$. □

Theorem 9. *For a given tree $T = (V, E)$, IS $S \subseteq V$, and positive integer $d \geq 2$, Algorithm 1 will find a (d)-MRS R in $O(n)$ time.*

Proof. Computing $q(v)$ for all vertices $v \in V$ bottom up, and constructing set R using those values, can all clearly be done in $O(n)$ time.

Instead of proving that the algorithm returns a d-MRS, we will prove that it returns a special type of d-MRS called a d-SMRS. Let us assume that the algorithm does not work, then one or more counterexamples should exist for the algorithm. Let $C = (T = (V, E), S \subseteq V, d)$ be a counterexample for our algorithm. We choose C such that for any other counterexample $C' = (T' = (V', E'), S' \subseteq V', d)$, C must uphold one of these constraints:

1. the height of T is less than height of T' or

2. the height of T equals height of T' and $|V| \leq |V'|$.

As a consequence we have that any input $I = (T'' = (V'', E''), S'' \subseteq V'', d)$ cannot be a counterexample, if at least one of the following constraints is true:

1. the height of T is at least the height of T'' or

2. the height of T equals the height of T'' and $|V| > |V''|$.

Let $v = \text{root}(T)$. It is easy to check that v cannot be a leaf. If $v \in S$, then v would be assigned $q(v) = 0$, which means that the algorithm returns $R = \emptyset$. If $v \notin S$, then v would be assigned $q(v) = d - 1$, which means that

the algorithm again returns $R = \emptyset$. It is clear that for a graph consisting of only a single node does not require any vertices to be removed.

Assume v is a linkpoint and that u is the child of v. Let $T' = (V, E)$ be a tree such that root$(T') = u$ and $I = (T', S, d)$ be another input for our algorithm. Since the height of T' is less than the height of T, we know that I can be solved by our algorithm. We can call R the returned set for input C and R' the returned set for input I. Since the only difference between T and T' is their root, it is quite clear that R' is a d-MRS for both C and I. So if $R = R'$, then R must be a d-MRS as well. It is also clear that v cannot be an element of any d-MRS. So if $R = R'$, then R would be a d-MRS. As it turns out, we can show that $R = R'$.

Let $v \in S$ and therefore $q(v) = 0$ for both input C and I, then we have that all nodes except for u get assigned the same value for both C and I. This means that for any node $u \neq w \neq v$ such that $w \in R'$, we have that $w \in R$. Since $v \in S$ we know from Lemma 5 that any d-MRS is also a d-SMRS. To show that R is a d-SMRS, we only have to check that $u \in R$ if and only if $u \in R'$:

- If $q(u) = 0$ for input C, then $u, v \in S$, which contradicts the fact that S has to be an IS.
- If $q(u) < d - 1$ for input C, then u must have a child w such that $q(u) = q(w) + 1$ for input C. For input I we should have the same assigned value to w. This means that $q(u) = d$ for input I, because $q(v) + q(w) + 2 < d$. This means that $u \in R'$ and we also have that $u \in R$.
- If $q(u) = d - 1$ for input C, then according to Lemma 4 that means that u has no (d, S)-descendant. However, for input I we know that v is a child of u, which leads to $q(u) = 1 < d$. Even though $q(u)$ is different for the two inputs, we still have that $u \notin R$ and $u \notin R'$.
- If $q(u) = d$ for input C, then u must have two children u_1 and u_2 such that $q(u_1) + q(u_2) + 2 < d$. The same is true for input I, which leads to u also being assigned value $q(u) = d$. So we have that $u \in R'$ and $u \in R$.

Let $v \notin S$ and therefore $q(v) = d-1$ for input I, then we have that all nodes except for u and v get assigned the same value for both C and I.This means that for any node w such that $w \in R'$ and $u \neq w \neq v$, we have that $w \in R$.

We know that $v \notin R$ because it is the root node and a linkpoint for input C. We also know that $v \notin R'$, because v is a leaf for input I. To show that R is a d-SMRS, we have to check that $u \in R$ if and only if $u \in R'$ and that $d_{T[V \setminus R]}(v, S)$ is maximized:

- If $q(u) = 0$ for input C, then that means that $u \in S$, which means that $q(u) = 0$ for input I as well. This means that $u \notin R$ and $u \notin R'$.

- If $q(u) < d-1$ for input C, then u must have a child w such that $q(u) = q(w) + 1$ for input C, but no child w' such that $q(w) + q(w') + 2 < d$. For input I we should have the same assigned values to all children of u. Since $q(v) = d-1$ for input I, we know that $q(w'') + q(v) + 2 \geq d$ for any child w'' of u. This means that $q(u)$ computes to the same value for input I as for input C. We therefore have that $u \notin R'$ and $u \notin R$. We know that $d_{T[V \setminus R]}(u, S)$ is maximized and since u is the only child of v, we know that $d_{T[V \setminus R]}(v, S)$ is maximized as well.

- If $q(u) = d - 1$ for input C, then according to Lemma 4 that means that u has no (d, S)-descendant. Since $q(v) = d - 1$ for input I, we have that $q(u) = d - 1$ for input I as well. So $u \notin R$ and $u \notin R'$. We know that $d_{T[V \setminus R]}(u, S)$ is maximized and since u is the only child of v, we know that $d_{T[V \setminus R]}(v, S)$ is maximized as well.

- If $q(u) = d$ for input C, then u must have two children u_1 and u_2 such that $q(u_1) + q(u_2) + 2 < d$. The same is true for input I, which leads to u also being assigned value $q(u) = d$. So we have that $u \in R'$ and $u \in R$. Since u is the only child of v and $u \in R$, we know that $d_{T[V \setminus R]}(v, S) = \infty$.

Assume v is a branchpoint. If $v \in S$, then any d-MRS is also a d-SMRS according to Lemma 5. To find a d-MRS for C, we can split the problem into smaller problems. For each child $v_i \in C(v)$ let $I_i = (T_i, S_i, d)$ such that $V_i = V(\text{tree}_T(v_i)) \cup \{v\}$, $T_i = T[V_i]$ with $\text{root}(T_i) = v$, and $S_i = S \cap V_i$. We know that inputs I_i can all be solved by our algorithm, because the trees have the same height, but less vertices than T. We also know that the correct output for input C is $\bigcup_{v_i \in C(v)} R_i$, where R_i is the output of the algorithm for input I_i. Since for both C and any I_i we have that $v' \in V_i$ gets assigned the same value $q(v')$, we know that the output of the algorithm for input C must be $\bigcup_{v_i \in C(v)} R_i$.

If $v \notin S$, then let $v_1, v_2 \in C(v)$ such that v_1 is assigned the lowest value over all children and v'' the second lowest value. Let I_i, T_i, S_i, and R_i be defined

as before. For any $u \in V_i$ such that $u \neq v$, we have that $q(u)$ is the same for both input C and input I_i. So $u \in R_i$ if and only if $u \in R$. So we only have to show whether or not $v \in R$ and if that makes R a d-SMRS of C or not.

- If $q(v_1) + q(v_2) + 2 < d$, then $q(v) = d$ for input C. Therefore we know that $v \in R$ and that $R = \{v\} \cup \bigcup_{v_i \in C(v)} R_i$. Since R_1 is a d-SMRS and $v_1 \notin R_1$, then we know that $d_{T_1[V_1 \setminus R_1]}(v_1, S)$ is maximized. Since R_2 is a d-SMRS and $v_2 \notin R_2$, then we know that $d_{T_2[V_2 \setminus R_2]}(v_2, S)$ is maximized. This means that removing an additional vertex to separate the closest (d, S)-descendants of v_1 and v_2 is necessary. This and the fact that $v \in R$ lead to the the conclusion that R is a d-SMRS.

- If $q(v_1)+q(v_2)+2 \geq d$, then $q(v) \leq d-1$ for input C. Therefore we know that $v \notin R$ and that $R = \bigcup_{v_i \in C(v)} R_i$. The closest (d, S)-descendants of v are at least $q(v')$ and $q(v'')$ away. Since $q(v') + q(v'') + 2 \geq d$, we know that they are at least distance d apart from each other. This means that R is a d-MRS for input C. We also know that for any I_i that $d_{T_i[V_i \setminus R_i]}(\text{root}(T_i), S_i)$ is maximized, which means that for C that $d_{T[V \setminus R]}(\text{root}(T), S)$ is maximized. Thus R is also a d-SMRS.

For any type of root v and any assigned values to the child(ren) of v, we come to the conclusion that C cannot be a counterexample. Thus contradicting the assumption that a counterexample exists. □

Note that Algorithm 1 actually has a worst case running time of $O(\log(d)n)$, because we assign numbers that require $O(\log(d))$ worst case time to assign. Since d is a constant, we have omitted it from Theorem 9. It is worth noting though, considering that in Section 4 we mentioned a way to generate algorithms that have worst case complexity of $O(n)$ too. Of course those algorithms will most likely have a factor polynomial in d.

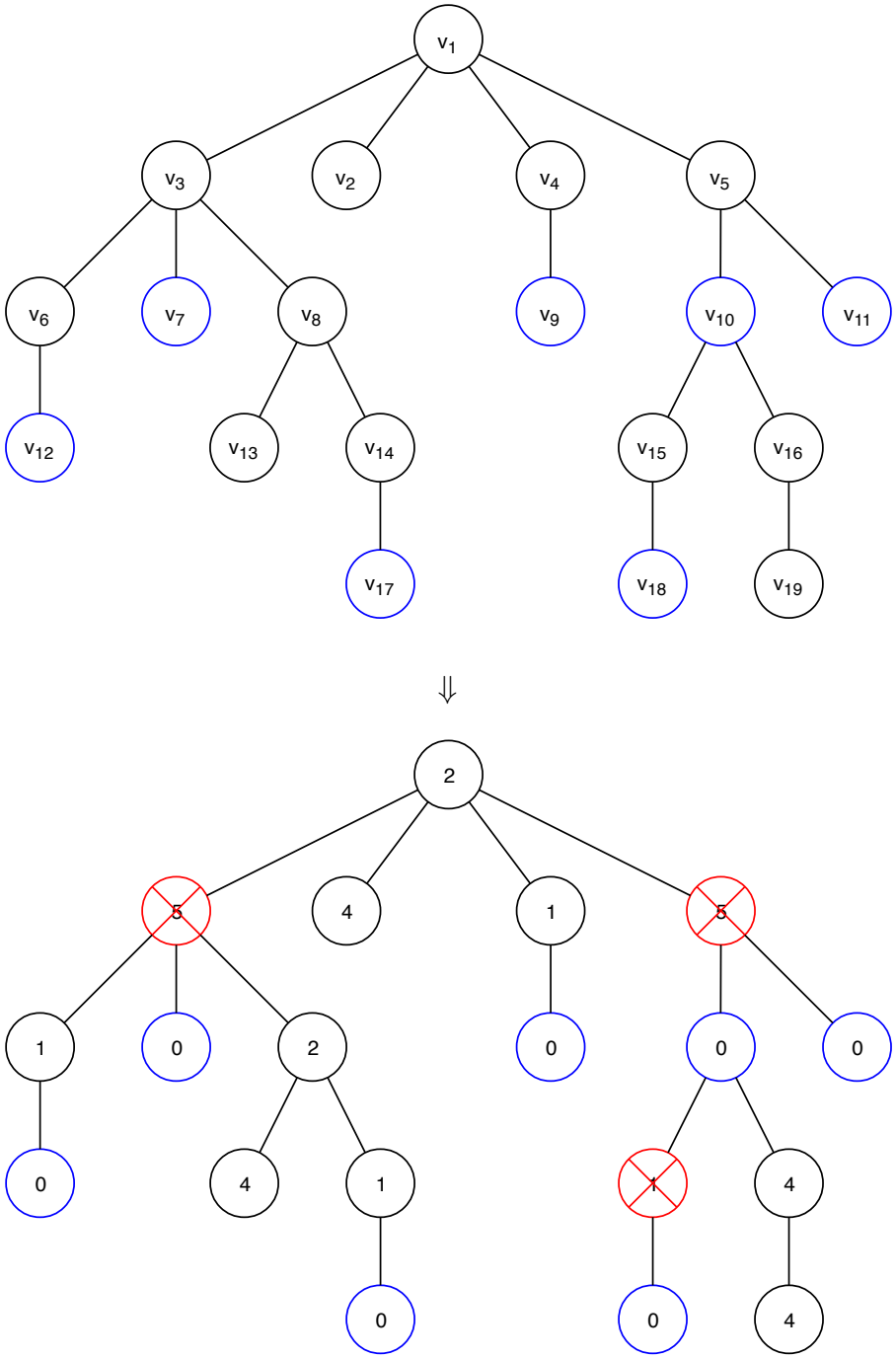

Figure 1: An example of a tree for which every node is assigned a value according to function q with input set $S = \{v_7, v_9, v_{10}, v_{11}, v_{12}, v_{17}, v_{18}\}$ (illustrated with a blue border) and $d = 5$. The 5-MRS found by our algorithm is $R = \{v_3, v_5, v_{15}\}$ (illustrated with a red cross).

6 The MRrDdIS Problem on Trees

Solving the MRrDdIS problem is somewhat more complicated, but it can be achieved in polynomial time using dynamic programming. For each node v in the tree we compute a (v, r, d)-MaxArray. By doing this bottom-up, we are able to calculate it for each node in polynomially time. A (v, r, d)-MaxArray can be seen as a 2-dimensional array of solutions for the MRrDdIS problem, but with additional constraints. One of these constraints is that we are only allowed to add vertices to our solution that are at least distance i away from v, such that i can range between 0 and $d - 1$. The other constraint is that we are allowed to remove at most j vertices, such that j can range between 0 and r. A more formal definition can be found in Section 2.

A depiction of a (v, r, d)-MaxArray can be seen in Figure 2. The top left corner, $M_{0\ 0}$, is a MDdIS of tree$_T(v)$. The top right corner, $M_{0\ r}$, is a MRrDdIS of tree$_T(v)$ and if v is the root of our tree T, then it is the solution to our problem. Each element $M_{0\ j}$ in the top row is a MRjDdIS of tree$_T(v)$. Cells that are more to the right are allowed to remove more vertices and cells more to the left less vertices. This is true for every row in the array. The height of a cell carries a different type of constraint. The lower a cell, the further all vertices in that cell have to be separated from v. So the top row, M_0, is the only row that allows v to be part of the solutions.

We solve the problem by splitting it into into three cases, such that the root node is either a leaf, linkpoint, or branchpoint. This can be seen in Algorithm 2.

$M_{0\,0}$	$M_{0\,1}$	...	$M_{0\,j}$	...	$M_{0\,r}$
$M_{1\,0}$	$M_{1\,1}$	...	$M_{1\,j}$	...	$M_{1\,r}$
...	...		...		...
$M_{i\,0}$	$M_{i\,1}$	...	$M_{i\,j}$	...	$M_{i\,r}$
...	...		...		...
$M_{d-1\,0}$	$M_{d-1\,1}$	...	$M_{d-1\,j}$	...	$M_{d-1\,r}$

Figure 2: An illustration of a (v, r, d)-MaxArray.

Algorithm 2: TMaxArray(T, r, d)

Input: Some subtree T and positive integers r and d.
Result: A $(\text{root}(T), r, d)$-MaxArray of T.

```
v ← root(T);
if |C(v)| = 0 then
    M ← LMaxArray(T, r, d);
end
else if |C(v)| = 1 then
    M ← LPMaxArray(T, r, d);
end
else
    M ← BPMaxArray(T, r, d);
end
return M;
```

Theorem 10. *For any tree $T = (V, E)$, $d \geq 2$, and $r \geq 0$ Algorithm 2 returns a $(root(T), r, d)$-MaxArray.*

Proof. We can prove that the algorithm works through induction.

1. If the root node of T is a leaf, then we call Algorithm 3, which according to Lemma 6 returns a $(\text{root}(T), r, d)$-MaxArray.

2. Assume that Algorithm 2 works for any subtree of T. If the root node of T is a linkpoint, then we call Algorithm 4. According to Lemma 7 it will return a $(\text{root}(T), r, d)$-MaxArray, because Algorithm 2 works for any subtree of T.

3. Assume that Algorithm 2 works for any subtree of T. If the root node of T is a branchpoint, then we call Algorithm 5. According to Lemma 8 it will return a $(\text{root}(T), r, d)$-MaxArray, because Algorithm 2 works for any subtree of T.

4. From base case 1 and induction steps 2 and 3 we can conclude using induction that Algorithm 2 will return a $(\text{root}(T), r, d)$-MaxArray.

□

6.1 Leaves

If root node v is a leaf, then it is trivial that we can just assign $\{v\}$ to all cells in the top row and $\emptyset$ to the others, as illustrated in Figure 3 and done in Algorithm 3.

Algorithm 3: LMaxArray(T, r, d)

```
Input: Tree T = ({v}, ∅), r and d are positive integers.
Result: A (v, r, d)-MaxArray of subtree T = ({v}, ∅).
for i ∈ [0, d − 1] do
    for j ∈ [0, r] do
        M_{i j} ← ∅;
        if i = 0 then
            M_{i j} ← {v};
        end
    end
end
return M;
```

Lemma 6. *For any tree $T = (\{v\}, \emptyset)$, $d \geq 2$, and $r \geq 0$ Algorithm 3 returns a (v, r, d)-MaxArray.*

Proof. Since T only consist of a single node, we know that any cell in the (v, r, d)-MaxArray either contains $\{v\}$ or $\emptyset$. Since we want to maximize each cell of or array we pick $\{v\}$ if it does not conflict with any constraints. Since the top row has no conflicting constraints, we assign set $\{v\}$ to the cells of that row. All other rows do conflict with this set, so we assign $\emptyset$ to all the other cells. □

{v}	{v}	...	{v}
∅	∅	...	∅
...	...		...
∅	∅	...	∅

Figure 3: The (v, r, d)-MaxArray of a leaf node v.

6.2 Linkpoints

If the root node v is a linkpoint, then there are three possible ways we can maximize the content of each cell in our array:

- Add v to the solution and pick from the child the largest solution set that is at least distance d away from v (bottom row).
- Remove v from the tree and pick from the child the largest solution set that has no distance restriction (top row), but with one less vertex to remove (one column to the left).
- Just pick from the child the largest solution set that is sufficiently far away from v (one row up).

In Figure 4 we see how each cell contains up to three different sets, generated from above strategies. Note that the first strategy can only be performed on cells in the top row, because all other rows do not allow v to be part of the solution. The second strategy cannot be applied to cells in the left column, because we are not allowed to remove nodes in the left column. The third strategy works different for the top row compared to all other rows. In the other rows we simply pick $M_{i-1\ j}$ for each cell $M'_{i\ j}$ of v. For the top row this is not possible. The largest set we can pick for $M'_{0\ j}$ is $M_{0\ j}$.

The left linkpoint of Figure 4 shows all possible strategies for each cell. As a result we see in the blue and yellow areas some solutions that are at most as large as other solutions. To be more precise, we know that for any $1 \leq j \leq r$ that $|M_{0\ j-1}| \leq |M_{0\ j}|$, which means that $M_{0\ j-1}$ can therefore be removed from consideration, as shown in the right linkpoint of Figure 4. The actual solution of each cell is the largest set of the ones we see in each cell. For each cell in the green area there is only one possible solution.

Now that we know which solutions we can consider for each cell, we can make an algorithm that actually fills the array. In Algorithm 4 we try to do this in such a way that the algorithm stays as compact and readable as possible. To achieve this, we take advantage of some, but not all, of the optimizations of the right linkpoint in Figure 4. In the first for-loop of the algorithm, we fill all rows except the top row by considering all solutions we see in the left linkpoint in Figure 4. After the first for-loop ended, we fill the top row, while only considering the solutions we see in the right linkpoint in Figure 4.

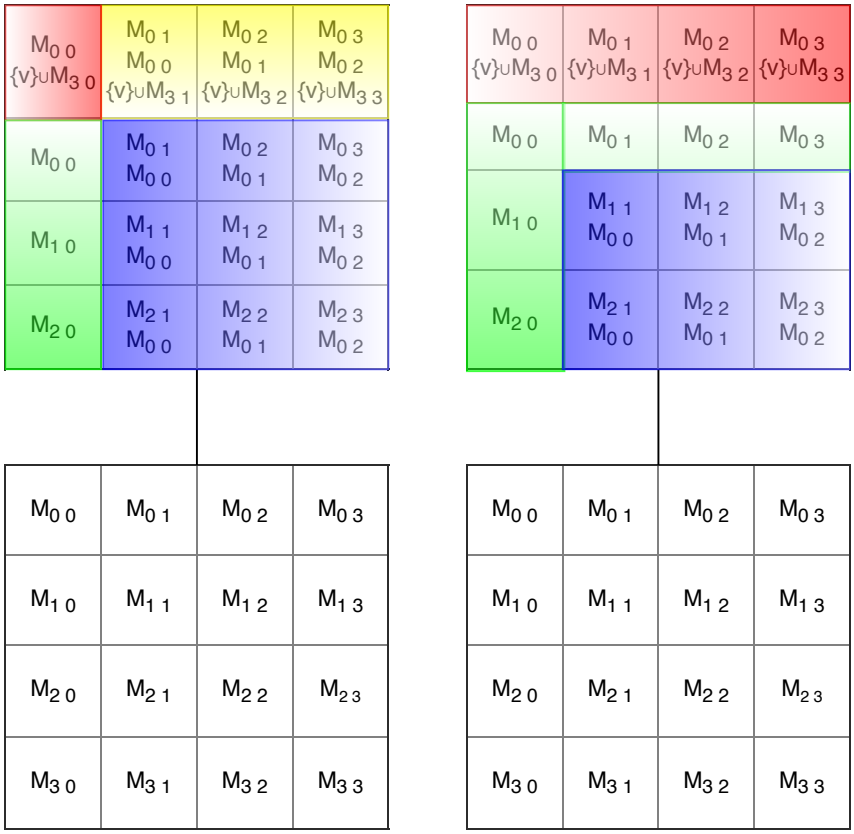

Figure 4: On the top we see the same linkpoint twice with its child on the bottom. The (v, r, d)-MaxArray of the linkpoint gets computed by picking the largest set in each cell. In the left linkpoint we see four areas coloured from left to right, top to bottom: red , yellow , green , blue . Cells in the red or blue area contain two sets to choose from, cells in the yellow area can choose out of three different sets, but cells in the green area are always assigned the same set. The yellow and blue areas in the left contain cells with sets that do not need to be considered and are therefore removed at the right. As a result there is no yellow area and both the green and red areas expanded to the right.

Algorithm 4: LPMaxArray(T, r, d)

Input: Some subtree T (with a linkpoint as the root) and positive integers r and d.
Result: A $(\text{root}(T), r, d)$-MaxArray of T.

```
v ← root(T);
A ← TMaxArray(tree_T(v_1), r, d);
for i ∈ [1, d − 1] do
    M_i ← A_{i−1};
    for j ∈ [1, r] do
        if |M_{i j}| < |A_{0 j−1}| then
            M_{i j} ← A_{0 j−1};
        end
    end
end
M_0 ← M_1;
for j ∈ [0, r] do
    if |M_{0 j}| < |A_{d−1 j} ∪ {v}| then
        M_{0 j} ← A_{d−1 j} ∪ {v};
    end
end
return M;
```

Lemma 7. *For any tree $T = (V, E)$ with $v = root(T)$, $d \geq 2$, and $r \geq 0$ Algorithm 4 returns a (v, r, d)-MaxArray, if Algorithm 2 returns a (v_1, r, d)-MaxArray for inputs $T' = tree_T(v_1)$, d, and r, where v_1 is the only child of v.*

Proof. Algorithm 4 constructs M using a (v_1, r, d)-MaxArray A. The algorithm assigns $A_{i-1\ 0}$ to $M_{i\ 0}$ for any $0 < i < d$. Since $A_{i-1\ 0}$ is a $(v_1, i-1, 0, d)$-MaxCell it is also a $(v, i, 0, d)$-MaxCell, which is what we want $M_{i\ 0}$ to be.

The algorithm assigns either $A_{i-1\ j}$ or $A_{0\ j-1}$ to $M_{i\ j}$ for any $0 < i < d$ and $0 < j \leq r$. Obviously $A_{i-1\ j}$ is a potential candidate of being a (v, i, j, d)-MaxCell. If we decide to add v to the removal set corresponding to $A_{0\ j-1}$, then $A_{0\ j-1}$ is a potential candidate as well. So to show that one of them has to be a (v, i, j, d)-MaxCell, we have to prove that no (v, i, j, d)-MaxCell exists that is larger than either one of them. Assume a larger (v, i, j, d)-Cell exists and it does not require us to add v to the corresponding removal set, then it would also have been a $(v_1, i-1, j, d)$-Cell greater than $A_{i-1\ j}$, which is in contradiction with our assumption that $A_{i-1\ j}$ is a $(v_1, i-1, j, d)$-MaxCell. Assume a larger (v, i, j, d)-Cell exists, but it requires us to add v to the

corresponding removal set, then it would also have been a $(v_1, 0, j-1, d)$-Cell greater than $A_{0\ j-1}$, which is in contradiction with our assumption that $A_{0\ j-1}$ is a $(v_1, 0, j-1, d)$-MaxCell. This means that either $A_{i-1\ j}$ or $A_{0\ j-1}$ is a (v, i, j, d)-MaxCell.

The algorithm assigns either $M_{1\ j}$ $(= A_{0\ j}$, because $|A_{0\ j}| \geq |A_{0\ j-1}|)$ or $A_{d-1\ j} \cup \{v\}$ to $M_{0\ j}$ for any $0 \leq j \leq r$. The two sets are both clearly $(v, 0, j, d)$-Cells. We use the same type of proof as before, to prove that at least one of the two is also a $(v, 0, j, d)$-MaxCell. Assume that a larger $(v, 0, j, d)$-Cell exists and it contains v. If we remove v, then we get a $(v_1, d-1, j, d)$-Cell greater than $A_{d-1\ j}$, which is in contradiction with our assumption that $A_{d-1\ j}$ is a $(v_1, d-1, j, d)$-MaxCell. Assume that a larger $(v, 0, j, d)$-Cell exists and it requires us to add v to the corresponding removal set, then it would also be a $(v_1, 0, j-1, d)$-Cell greater than $A_{0\ j-1}$, which is in contradiction with our assumption that $A_{0\ j-1}$ is a $(v_1, 0, j-1, d)$-MaxCell. Assume that a larger $(v, 0, j, d)$-Cell exists and that does not contain v or add it to the corresponding removal set. Such a set would also be a $(v_1, 0, j, d)$-Cell greater than $A_{0\ j}$, which is in contradiction with our assumption that $A_{0\ j}$ is a $(v_1, 0, j, d)$-MaxCell. This all means that either $A_{0\ j}$ or $A_{d-1\ j} \cup \{v\}$ is a (v, i, j, d)-MaxCell.

□

6.3 Branchpoints

If v is a branchpoint, then computing the (v, r, d)-MaxArray becomes a bit more complicated. Nevertheless we can apply similar strategies to it like we did for the linkpoint. The first two strategies, where we either add v to the solution or remove it from the tree, are mostly the same as with the linkpoint. As Figure 5 illustrates, if we want to add v to the solution, then we cannot combine it with solution sets that are less than distance d away from v. This means that we can only pick solutions from the bottom rows of the children. Figure 5 also illustrates that when we remove v, the opposite is true. When v gets removed, we could pick solutions from any row in the children, but if we want to maximize our solution, we should choose from the top row of each child.

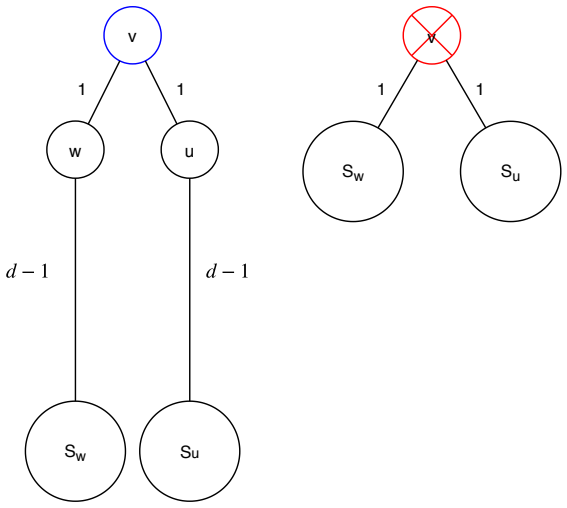

Figure 5: Left we see a branchpoint that we decide to add to our solution. As a result all child solution sets, S_u and S_w, that we want to combine with v to create a new solution, have to be at least distance d away from v. At the right we see a branchpoint that gets removed from the tree. As a result we can combine child solution sets that even contain the children themselves.

In Figure 6 we see an example where v has $k \geq 3$ children. Nodes S_1 to S_k each represent a solution for v_1 to v_k. We want to combine these solutions to create a new solution for v. Since we want the new solution to be a (v, i, j, r, d)-MaxCell we have to keep a few things into account:

- We want to maximize the solution of v, so we should choose solutions of the child nodes that are as large as possible.
- The child solutions must all be at least distance i away from v.
- The child solutions must be at least distance d apart from each other.
- The combined number of vertices that needs to be removed cannot be more than j.

For now we will only focus on the first three points. The height of a solution node in Figure 6 represents the maximum height of its highest element. If a solution node is lower, then that means it is more restricted and possibly smaller than when its higher and closer to v. So to maximize v we want the solution nodes to be as high as possible. However, they must remain distance d apart from each other, so they cannot all be high up in the tree. They also

must remain distance i away from v. In Figure 6 we see two of the $O(d)$ configurations that satisfy our criteria. If we start with the left configuration and lower S_1, then we can raise S_2 to S_k with the same amount. As a result S_1 may decrease in size, but S_2 to S_k might increase in size. So the combined solution could both increase and decrease. Once we have reached the configuration on the right, we must stop or else solutions S_2 to S_k would get to close to each other. All the remaining configurations that satisfy our criteria can be generated by repeating the process, but with swapped heights. This means that we have to check $O(dk)$ configurations.

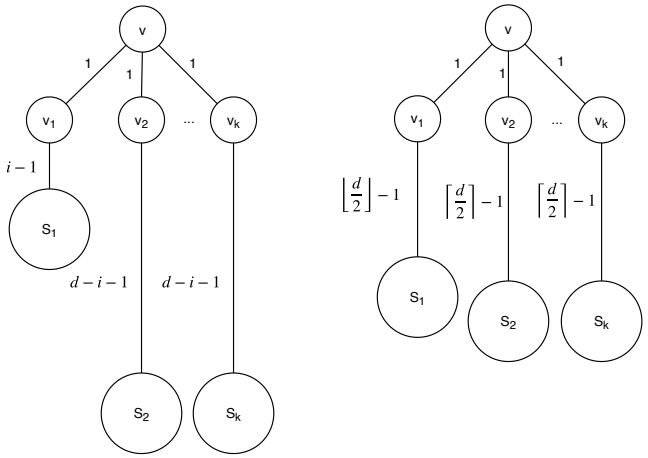

Figure 6: A branchpoint v, its child nodes v_1 to v_k and their respective solution sets S_1 to S_k. Algorithm 5 starts with the configuration on the left and ends with the configuration on the right.

In Figure 7 we see an example where v has only two children. We could treat this situation the same way as when v has $k \geq 3$ children. However, when we reach the configuration at the right of Figure 7, we could continue instead of stopping. If we continue S_w and S_u will still be distance d apart from each other. If we decide to stop and then repeat with swapped heights, then we would simply generate the same configurations as when we would have continued, but in opposite order. So we can treat a branchpoint v with 2 children in the same way as a branchpoint with $k \geq 3$ children.

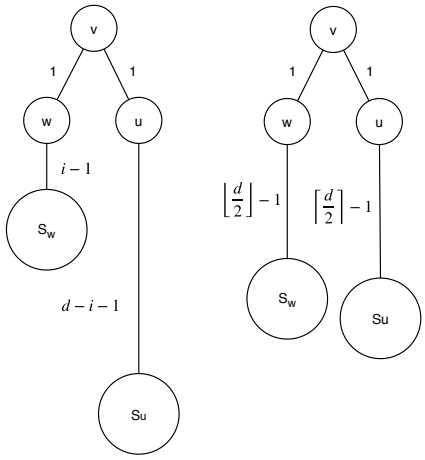

Figure 7: A branchpoint v, its child nodes w to u and their respective solution sets S_w to S_u. Algorithm 5 starts with the configuration on the left and ends with the configuration on the right.

So far we have treated the child solutions of a branchpoint as if they were part of a pulley system, but note that this does not always have to be the case. In Figure 8 we see a situation in which all child nodes are forced to be so far down, that they always remain at least distance d apart from each other. In that case we only have to check the configuration shown in Figure 8.

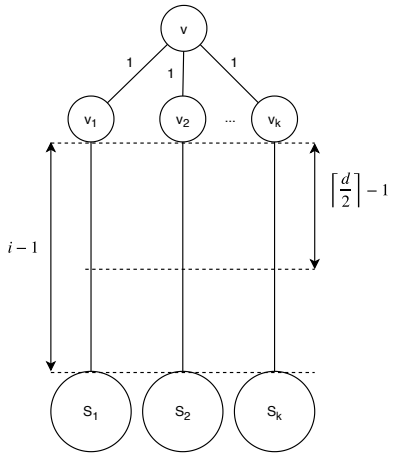

Figure 8: A branchpoint v, its child nodes v_1 to v_k and their respective solution sets S_1 to S_k. This is the only configuration Algorithm 5 checks when $i \geq \lceil \frac{d}{2} \rceil$.

Algorithm 5: BPMaxArray(T, r, d)

Input: Some subtree T (with a branchpoint as the root) and positive integers r and d.
Result: A $(\text{root}(T), r, d)$-MaxArray of T.

```
v ← root(T);
for u ∈ C(v) do
    P_u ← TMaxArray(tree_T(u), r, d);
end
M ← P_{v_1};
for i ∈ [⌈d/2⌉, d − 1] do
    lowrows ← ∅;
    for u ∈ C(v) do
        lowrows ← lowrows ∪{P_{u i−1}};
    end
    M_i ← ORSD(lowrows, r);
end
for w ∈ C(v) do
    for i ∈ [1, ⌈d/2⌉ − 1] do
        for k ∈ [i − 1, ⌊d/2⌋ − 1] do
            highrows ← {P_{w k}};
            for u ∈ C(v) do
                if u ≠ w then
                    highrows ← highrows ∪{P_{u d−k−2}};
                end
            end
            S ← ORSD(highrows, r);
            for j ∈ [0, r] do
                if |M_{i j}| < |S_j| then
                    M_{i j} ← S_j;
                end
            end
        end
    end
end
```

```
toprows ← ∅;
for u ∈ C(v) do
    toprows ← toprows ∪{P_{u 0}};
end
X ← ORSD(toprows, r);
for i ∈ [1, d − 1] do
    for j ∈ [1, r] do
        if |M_{i j}| < |X_{j−1}| then
            M_{i j} ← X_{j−1};
        end
    end
end
M_0 ← M_1;
bottomrows ← ∅;
for u ∈ C(v) do
    bottomrows ← bottomrows ∪{P_{u d−1}};
end
O ← ORSD(bottomrows, r);
for j ∈ [0, r] do
    if |M_{0 j}| < |O_j ∪ {v}| then
        M_{0 j} ← O_j ∪ {v};
    end
end
return M;
```

Lemma 8. *For any tree $T = (V, E)$ with $v = root(T)$, $d \geq 2$, and $r \geq 0$ Algorithm 5 returns a (v, r, d)-MaxArray, if Algorithm 2 returns a (v', r, d)-MaxArray for inputs $T' = tree_T(v')$, d, and r, for all $v' \in C(v)$.*

Proof. In the algorithm we see that the first for-loop stores the (v', r, d)-MaxArray of each child v'. After that M is assigned one of the arrays, to ensure that M contains valid solutions (besides them not being the maximum) to compare against later on.
In the second for-loop we explore possible solutions for each (v, i, j, d)-MaxCell with $\lceil \frac{d}{2} \rceil \leq i \leq d - 1$ and $0 \leq j \leq r$. None of these solutions contain v or require v to be removed. Each solution is created by combining the $(v', i - 1, r, d)$-MaxRows of each child v' using Algorithm 6. We know that since $i \geq \lceil \frac{d}{2} \rceil$ that the solutions of the children will be at least distance $i - 1 + i - 1 + 2 \geq 2 \cdot \lceil \frac{d}{2} \rceil \geq d$. We also know that we cannot pick solutions higher in the array to create a larger solution, because either the solution

does not become larger or it contains elements too close to v. We know from Lemma 11 that Algorithm 6 will combine them in the most optimal way. Since we only assign a single solution to each cell and the previous values were just meant as place holders, we can assign the results without checking for improvements.
In the third for-loop we explore possible solutions for each (v, i, j, d)-MaxCell with $1 \leq i \leq \lceil \frac{d}{2} \rceil - 1$ and $0 \leq j \leq r$. None of these solutions contain v or require v to be removed. For one child w we select a (w, k, r, d)-Row, while we select for each other child u a $(u, d-k-2, r, d)$-Row. Just like before we use the ORSD algorithm to combine them optimally. We do this for all $i - 1 \leq k \leq \lfloor \frac{d}{2} \rfloor - 1$. Since $k \geq i - 1$ we ensure no elements will be too close to v and $k \leq \lfloor \frac{d}{2} \rfloor - 1$ ensures that we do not create solutions with two elements that are to close together. This can be shown as follows:

- The distance between two elements of the children for which height $d-k-2$ was chosen is at least $d-k-2+d-k-2+2 \geq$ $d + (d - 2 \cdot (\lfloor \frac{d}{2} \rfloor - 1)) - 2 = d + (d - 2 \cdot \lfloor \frac{d}{2} \rfloor) \geq d$.

- The distance between an element of the child with height k and an element of a child with height $d-k-2$ is at least $k+d-k-2+2 = d$.

Since we are exploring multiple solutions per cell, we have to check for each cell if a new solution improves the old one, unlike before.
The fourth and fifth for-loops explore solutions for which the root node gets removed. They explore these possible solutions for each (v, i, j, d)-MaxCell with $1 \leq i \leq d$ and $1 \leq j \leq r$. Since we remove v, we can guarantee that solutions of the children will not be too close to each other or to v, whatever we choose. This means that we should pick solutions the top rows from each child and combine them with the ORSD algorithm. The resulting row of solutions gets stored in X. Since we removed a node, we have to assign the $j-1$-th element of X to the j-th cell of each row.
So far we have yet to explore actual solutions for the top row. As it turns out, we can just assign everything from the second highest row to the top row. When we explored the solutions were v was removed, we were able to treat all rows exactly the same, so we can treat the top row the same too. The solutions where v is not removed from the tree or added to the solution are a different case. This is because we cannot choose rows higher than the top row from the child arrays. As a result we have that the best solutions for the top row are the same as for the second row. Except for solutions that include v. The last for-loop explores this case. If we add v to the solution, then we must keep the child soltuions distance d away from v. This means we can only combine the bottom rows with v.

After the algorithm has finished, several possible solutions have been explored for each cell $M_{i\ j}$. For $i = 0$ and $j > 0$, we have explored all possible solutions except the ones where in the second for-loop, since they would have broken the distance constraints. For $i = 0$ and $j = 0$, we have explored the same solutions, except the ones were v (or any other node) gets removed. For $1 \leq i \leq \lceil \frac{d}{2} \rceil - 1$ and $j = 0$ we have explored the same solutions as for $i = 0$ and $j = 0$, except the ones were v was included in the solution. For $1 \leq i \leq \lceil \frac{d}{2} \rceil - 1$ and $j > 0$ we have explored the same solutions as for $i = 0$ and $j > 0$, except the ones were v was included in the solution. For $\lceil \frac{d}{2} \rceil \leq i \leq d-1$ and $j = 0$, we have explored only the solutions of the second for-loop. Exploring the third for-loop was unnecessary, since the solutions would have been smaller than the ones we already found. The other for-loops would explore solutions were we either removed v from the tree or added v to the solution, which would not have been valid solutions. For $\lceil \frac{d}{2} \rceil \leq i \leq d-1$ and $j > 0$, we have explored the same solutions, but also the ones were v gets removed from the tree. □

We assume that from the definition of a (v, r, d)-MaxArray and the ideas behind the algorithm, that it is clear how we try to solve the problem and why it is correct. That is why we prove the correctness of Algorithm 5 by listing for each cell in the array of branchpoint v, which solutions are explored and which are not. By showing that the algorithm works as intended, we hope to convince the reader the correctnes of the algorithm. A formal proof would have had to check a lot of different case, considering that there are 4 different types of solutions we explore and the fact that there are 6 groups of cells such that cells from different groups explore a different subset of solutions.

6.4 Distributing the removal set

We have seen that for a branchpoint with k children there are $O(dk)$ height configurations that need to be checked. Besides deciding height configurations, we also need to decide how to distribute the amount of vertices we are allowed to remove among the children. To do this we use Algorithm 6. As input it takes a row from each child and as output it generates a row for v. The way it works is illustrated in Figure 9 and Figure 10. Note that the rows in the figures do not contain sets of vertices, but set sizes. So instead of combining two sets, we add two numbers together.

When we combine rows with each other, we have to keep in mind that solutions more to the right of a row are always bigger, but also require more

vertices to remove. So if we are not allowed to remove any vertices then we can only combine the leftmost elements of all input rows. If we are allowed to remove j vertices, then there are $O(j^k)$ possible ways to distribute j among the k children. Fortunately, we can use dynamic programming to solve this problem in polynomial time as well. Instead of finding the ideal distribution for k children, we start by finding the ideal way to do so for 2 children. Once we have done this, we get a new row.

Figure 10 illustrates how a new row is created. The $(j+1)$-th cell from the left of the output row is created by combining the $(j'+1)$-th cell from the left of one row with the $(j-j'+1)$-th cell from the left of the other row, for some $0 \leq j' \leq j \leq r$. We choose j' to maximize $(j+1)$-th cell. Note that in Algorithm 6 we do not bother to fill a whole array with numbers, but just iterate through all the possible combinations illustrated by Figure 10 and pick the best.

Having found a whole row of distributions for two children opens the door for distributions with three children. In fact if we have computed a whole row of distributions for $k-1$ children we can easily compute such a row for k children. We just combine our computed row with an input row that we have not used yet. This is exactly what we do in Algorithm 6 and what we illustrate with Figure 9.

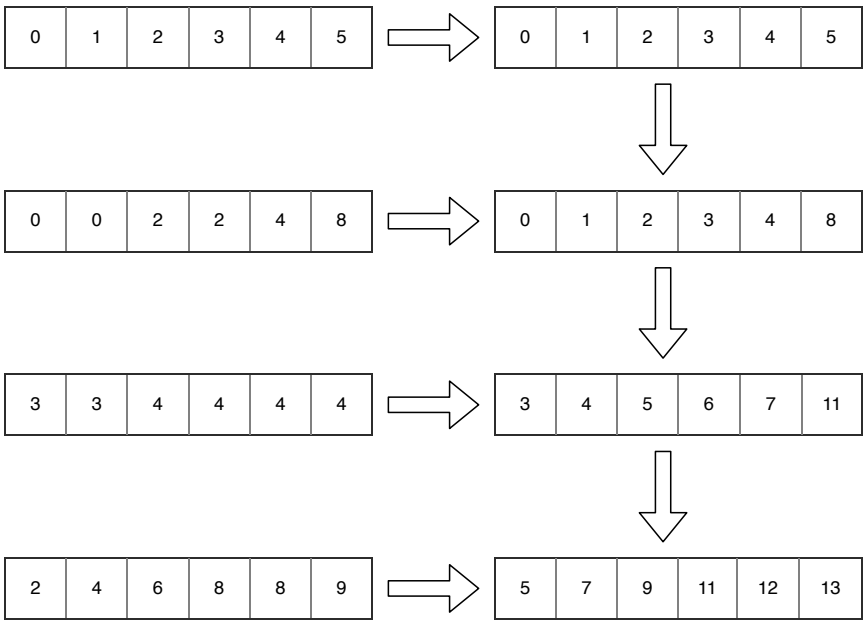

Figure 9: An example of how Algorithm 6 works. On the left we see a representation of a (v, p, r, d)-MaxRowSet for $r = 5$ and some v, p, and d. Each row comes from a (v, r, d)-MaxArray of a child of v. Each cell contains the size of a solution instead of the solution itself. On the right we see the largest set sizes you can create by combing an input row and an earlier created row. On the bottom right is the final result.

	3	4	5	6	7	11
2	(5)	6	7	8	9	13
4	(7)	8	9	10	11	-
6	(9)	10	11	12	-	-
8	(11)	(12)	(13)	-	-	-
8	11	12	-	-	-	-
9	12	-	-	-	-	-

Figure 10: An example of how Algorithm 6 computes the final row of Figure 9. On the top and left we see the two input rows. Each cell in the table contains the sum of an element of each row. The circled numbers are the highest numbers in their diagonal and are therefore picked by the algorithm.

Algorithm 6: ORSD(A, r)

Input: Input A is a (v, p, r, d)-MaxRowSet for some vertex v and integers p and d. Input r is some positive integer.
Result: A SemiMaxRow of A.

```
M ← A_1;
for i ∈ [2, |A|] do
    M' ← M;
    for j ∈ [1, r] do
        for k ∈ [0, j] do
            if |M_j| < |M'_k ∪ A_{i, j−k}| then
                M_j ← M'_k ∪ A_{i, j−k};
            end
        end
    end
end
return M;
```

Lemma 9. *For any (v, p, r, d)-MaxRowSet A and $r \geq 0$, such that v is some vertex and $0 \leq p$ and $d \geq 2$ are integers, Algorithm 6 returns a SemiMaxRow of A.*

Proof. We can prove that at the end of each iteration of the for-loop that M is a SemiMaxRow of $A^i = \{A_j \in A | 0 \leq j \leq i\}$ using induction:

1. When $M = A_1$, then M is a SemiMaxRow of A^1, because:
   - A^1 is clearly a (v, p, r, d)-MaxRowSet.
   - Since $|A^1| = 1$, we know that $\forall 0 \leq j \leq r$ that $|A_{1\ j}| \geq |S|$ such that $S = A^1_{1\ q_1} = A_{1\ q_1}$ with $0 \leq q_1 \leq j$.
2. Let $2 \leq i \leq |A|$ Assume that M' is a SemiMaxRow of A^{i-1}. As a result we get $\forall 0 \leq j \leq r$ that M_j is the maximum sized element of $\{M'_k \cup A_{i\ j-k} | 0 \leq k \leq j\}$. We know that M must be a SemiMaxRow of A^i, because:
   - A^i is clearly a (v, p, r, d)-MaxRowSet.
   - Let $S = \bigcup_{k=1}^{i} A_{k\ q_k}$ for some $0 \leq j \leq r$ such that $0 \leq \sum_{k=1}^{i} q_k \leq j$. Let $S' = \bigcup_{k=1}^{i-1} A_{k\ q_k}$ and $0 \leq \sum_{k=1}^{i-1} q_k = j' \leq j$, since M' is a SemiMaxRow

of A^{i-1}, we know that $|M'_{j'}| \geq |S'|$. This would mean that:

$$|M_j| \geq |M'_{j'}| + |A_{i\ j-j'}| \geq |S'| + |A_{i\ j-j'}| \geq |S'| + |A_{i\ q_i}| = |S|.$$

The last inequality is true, because $j' + q_i \leq j$, which means that $|A_{i\ j-j'}| \geq |A_{i\ q_i}|$.

3. From base case 1 and induction step 2 we can conclude using induction that M is a SemiMaxRow of $A^i = \{A_j \in A | 0 \leq j \leq i\}$ at the end of each iteration of the for-loop.

□

6.5 Complexity analysis

Now that we have validated the algorithms for finding a MRrDdIS, we can measure their complexity.

Lemma 10. *Algorithm 6 runs in $O(kr^2n)$ worst case time for which $k = |A|$.*

Proof. Algorithm 6 iterates over $k-1$ elements of A during which the algorithm iterates another $O(r^2)$ times. Over some of those $O(r^2k)$ iterations, the algorithm will assign a set of worst case size $O(n)$. This means that the algorithm runs in $O(kr^2n)$ worst case time. □

Theorem 11. *Algorithm 2 runs in $O(d^2r^2n^3)$ worst case time.*

Proof. We can prove the above statement using induction:

1. If we call Algorithm 3, then the running time will be $O(dr)$.

2. Algorithm 4 has a for-loop that iterates d times over r iterations in which an assignment of worst case size $O(n)$ is done. So the whole for-loop has a worst case of $O(drn)$ time. The other for-loop clearly requires only $O(rn)$ worst case time. Besides those two for-loops, the algorithm also calls Algorithm 2 for a tree with one node less. If we assume the above statement is correct for smaller trees, then Algorithm 4 also has a worst case running time of $O(d^2r^2n^3)$.

3. Algorithm 5 has multiple for-loops, but the third one clearly takes the longest time. The for-loop iterates $O(d^2k)$ times such that k is the number of children of v. Each iteration it performs a series of tasks of which two for-loops and a call to Algorithm 6 clearly take the most

time. The first for-loop takes $O(rn)$ time, because each iteration we append $O(r)$ set of worst case size $O(n_u)$ such that $u \in V(c)$ and n_u is the amount of nodes in $\text{tree}_T(u)$. Since $\sum_{u \in C(v)} n_u = n$ we have that the for-loop takes $O(rn)$ worst case time. The other for-loop also takes $O(rn)$ running time if we assume the assignments take $O(n)$ time over $O(r)$ iterations. According to Lemma 10, it will take $O(kr^2n)$ time to run Algorithm 6. So in the worst case the for-loop would run in $O(d^2r^2n^3)$ when $k = O(n)$ and one of the children of v has an array of which $O(rd)$ cells have $O(n)$ sized solutions.
Besides all those for-loops, the algorithm also calls Algorithm 2 for all its children. If we assume the above statement is correct for smaller trees, then we have that $\sum_{u \in C(v)} O(d^2r^2n_u^3) \leq O(d^2r^2n^3)$. So Algorithm 5 also has a worst case running time of $O(d^2r^2n^3)$.

4. From base cases 1 and 2 and induction step 3 we can conclude using induction that the algorithm runs in $O(d^2r^2n^3)$ time in the worst case.

□

Note that d is a constant, which means that we can omit it to get a worst case running time of $O(r^2n^3)$. We included d in our analysis to show that this algorithm is polynomial even if d were variable. It is clear that Algorithm 2 is not very efficient. If we consider that r is a variable that can be of size $O(n)$ in the worst case, then that would mean that our algorithm takes $O(n^5)$ in the worst case.

The ORSD algorithm is the main reason for the inefficiency of our algorithm. So we either have to call it less or make it more efficient. Calling it less might be difficult, but it can significantly affect the efficiency. From each child we choose the same row $O(kd)$ (k is amount of children of the root node) times to combine. If we could do it only once instead, then that would save us a worst case factor of $O(n)$ time. We can achieve this by doing all needed computations with the ORSD algorithm in bulk, storing the results in an array, and then choose which results get assigned to which cells. We are not going into much detail on how to store and retrieve the results, but we will explain how we can reduce the amount of calls and the cardinality of input A to a constant size.

For starters we want for each child, that the rows of certain height of all other children are combined. An easy way to achieve this is by combing the rows of

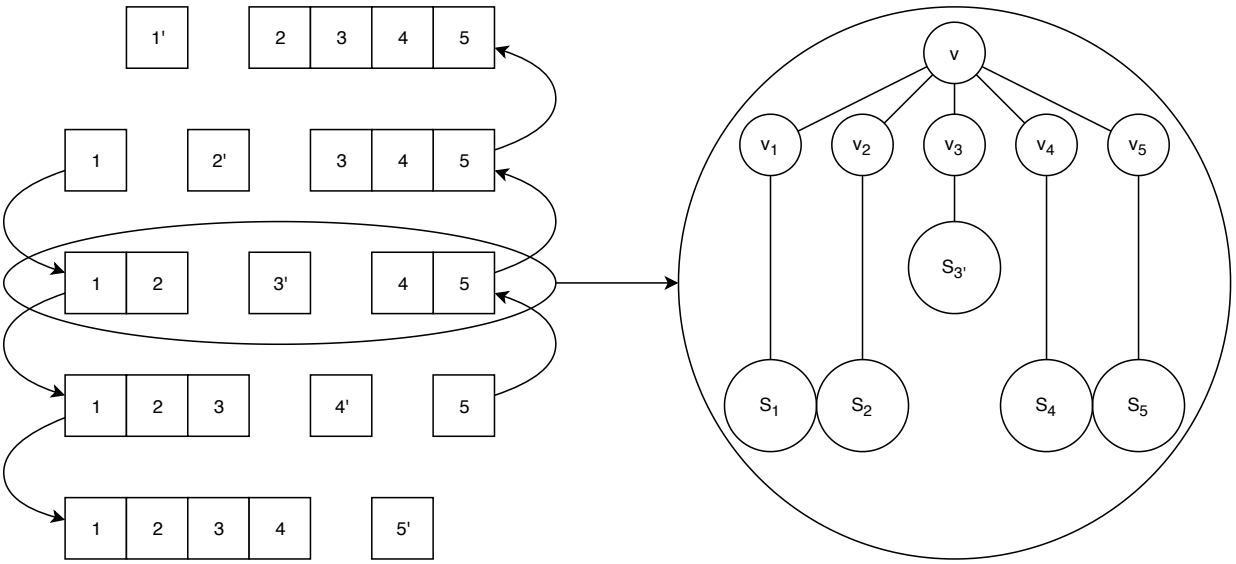

Figure 11: A more optimized approach of calling Algorithm 6. Each square on the left represents a row from a child. The number tells us which child. Note that the squares with the marked numbers represent higher rows from the same children. The arrows represent the use of the ORSD algorithm to combine squares into rectangles. The arrows going up and down add a single square to an earlier created solution. The arrow pointing to the right combines three solutions to find the optimal distribution of removal nodes for the configuration on the right.

the leftmost children in all possible $O(k)$ ways and the rightmost in all possible $O(k)$. Since we can do this iteratively, we would only need to combine two results at a time using the ORSD algorithm a total of $O(k)$ times. Figure 11 illustrates this at the left. This operation results in worst case running time of $O(kr^2n)$. Then to compute the actual results for child l, we take the result of the $l-1$ leftmost children, the l-th child, the $k-l$ rightmost children, and combine all three of them. We do this for $O(k)$ children and since $|A| = 3$ each call only takes $O(r^2n)$ time. On the left of Figure 11 we see how this is done for $l = 3$, to create the optimal distribution for the configuration on the right. This operation will also take $O(kr^2n)$ time in the worst case or $O(n^4)$ considering that both k and r can be of size $O(n)$ in the worst case. We only have ignored that there are $O(d^2)$ different configurations to explore, but since d is a constant, we can ignore this. So with these optimizations, we would probably get an algorithm of worst case $O(d^2r^2n^2)$ time.

Another idea to even further improve the algorithm is to reduce the amount

of assignments. We could do this by only assigning the largest set instead of assigning a set each time we find one that is bigger. We could also reduce the assignment operation itself by introducing a new data structure that instead of storing the whole set, stores references to the child solutions that are combined to make the whole set. This would change an assignment of worst case $O(n)$ time to one of $O(k+\log(n))$ time. This may not seem a large improvement since $O(k) = O(n)$ in the worst case. But note that the total amount of children of all nodes must be $n-1$, which would mean that an assignment of size $O(k+\log n)$ for each node would result in a total running time of $O(n + n\log n)$ instead of $O(n^2 + n\log n)$.

7 Conclusion

We have introduced a new problem called the MRrDdIS problem, which is a general. This problem has not been studied before to the best of our knowledge. We also introduced the d-MRS problem and the (r,d)-ORS problem, which are all related to the MRrDdIS problem. We have proven that the decision variants of the MRrDdIS problem and the d-MRS problem are both NP-complete for general graphs, but we also presented algorithms that solved both the MRrDdIS and the d-MRS problem in polynomial time for trees. In fact, we were even able to prove that the d-MRS problem can be solved in linear time for all graphs with bounded treewidth.

7.1 Summary of the results

In summary, what we have shown:

- The RrDdIS problem is NP-complete for general graphs.
- The (r,d)-RS problem is NP-complete for general graphs.
- The d-MRS problem can be solved in linear time on graphs with bounded treewidth.
- An algorithm solving the d-MRS problem in $O(n)$ time on trees.
- An algorithm solving the MRrDdIS problem in $O(d^2r^2n^3)$ time on trees.

7.2 Future research

We introduced the RrDdIS problem such that r is an input variable, but it might also be interesting to look into a variation where r is constant. It would

probably be not too hard to prove, using Courcelle's theorem, that this new problem can be solved in linear time on graphs with bounded treewidth.

While the algorithm we used to solve the MRrDdIS was not very efficient, it might be possible to modify it in such a way that it can be used on more general graphs. We could also look into optimizing the algorithm.

The (r, d)-RS problem was proven to be NP-complete, but for certain specific cases it is still unknown whether this is true. Namely if the given IS S has a size of 2 and distance $d \geq 5$. Even though it is solvable in polynomial time for $|S| = 2$ and $d = 4$, we expect the problem to be NP-complete for $d = 5$, but a proof has yet to be constructed.

While we also introduced the (r, d)-ORS problem, we did not proof it to be NP-hard or found any algorithms solving it. In the future, it might be worthwhile to focus on this problem on its own, considering that we could combine it with results from the MDdIS problem to solve the MRrDdIS problem. We could for starters look into approximation schemes on graph classes for which algorithms solving the MDdIS problem already exist.

References

[1] Geir Agnarsson, Peter Damaschke, and Magnús M. Halldórsson. Powers of geometric intersection graphs and dispersion algorithms. *Discrete Applied Mathematics*, 132(1-3):3–16, 2003.

[2] Gábor Bacsó, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Zsolt Tuza, and Erik Jan van Leeuwen. Subexponential-time algorithms for maximum independent set in P_t-free and broom-free graphs. *Algorithmica*, 81(2):421–438, 2019.

[3] Gábor Bacsó, Dániel Marx, and Zsolt Tuza. H-free graphs, independent sets, and subexponential-time algorithms. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63, pages 3:1–3:12, 2017.

[4] Binay K. Bhattacharya and Michael E. Houle. Generalized maximum independent sets for trees in subquadratic time. In *10th International Symposium on Algorithms and Computation (ISAAC 1999)*, pages 435–445. Springer, 1999.

[5] Richard B Borie, R Gary Parker, and Craig A Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(1-6):555–581, 1992.

[6] Richard B Borie, R Gary Parker, and Craig A Tovey. Solving problems on recursively constructed graphs. *ACM Computing Surveys (CSUR)*, 41(1):4, 2009.

[7] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[8] Erhan Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990.

[9] Hiroshi Eto, Fengrui Guo, and Eiji Miyano. Distance-d independent set problems for bipartite and chordal graphs. *Journal of Combinatorial Optimization*, 27(1):88–99, 2014.

[10] Hiroshi Eto, Takehiro Ito, Zhilong Liu, and Eiji Miyano. Approximability of the distance independent set problem on regular graphs and planar graphs. In *10th International Conference on Combinatorial Optimization and Applications (COCOA 2016)*, pages 270–284. Springer, 2016.

[11] Hiroshi Eto, Takehiro Ito, Zhilong Liu, and Eiji Miyano. Approximation algorithm for the distance-3 independent set problem on cubic graphs. In *11th International Workshop on Algorithms and Computation (WALCOM 2017)*, pages 228–240. Springer, 2017.

[12] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[13] Lars Jaffke, O-joung Kwon, Torstein J. F. Strømme, and Jan Arne Telle. Generalized distance domination problems and their complexity on graphs of bounded mim-width. In *13th International Symposium on Parameterized and Exact Computation (IPEC 2018)*, volume 115, pages 6:1–6:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[14] Sangram K. Jena, Ramesh K. Jallu, Gautam K. Das, and Subhas C. Nandy. The maximum distance-d independent set problem on unit disk graphs. In *12th International Workshop on Frontiers in Algorithmics (FAW 2018)*, pages 68–80. Springer, 2018.

[15] Ioannis Katsikarelis, Michael Lampis, and Vangelis Th. Paschos. Structurally parameterized d-scattered set. In *44th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2018)*, pages 292–305. Springer, 2018.

[16] Michael J. Kuby. Programming models for facility dispersion: The p-dispersion and maxisum dispersion problems. *Geographical Analysis*, 19(4):315–329, 1987.

[17] Pedro Montealegre and Ioan Todinca. On distance-d independent set and other problems in graphs with "few" minimal separators. In *42nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2016)*, pages 183–194. Springer, 2016.

[18] Daniel J. Rosenkrantz, Giri K. Tayi, and S. S. Ravi. Facility dispersion problems under capacity and cost constraints. *Journal of Combinatorial Optimization*, 4(1):7–33, 2000.

[19] Volker Turau and Sven Köhler. A distributed algorithm for minimum distance-k domination in trees. *Journal of Graph Algorithms and Applications*, 19(1):223–242, 2015.