Exact Exponential-Time and Treewidth-Based Algorithms for Defence-like Domination Problems

Mats Veldhuizen
Utrecht University
ICA-3971740

July 7, 2020

Abstract

Defence-like domination problems are variants to Dominating Set, where the goal is to defend a graph against attacks using guards. This is done by moving guards along edges in our graph in a response to these attacks. In this thesis we will define our own general definition of defence-like domination problems and present exact exponential-time and treewidth-based algorithms for a selection of these problems. Specifically we present an $\mathcal{O}^*(4^n)$ time algorithm for k-Turn Defensive Domination, which is a problem newly introduced in this thesis. Next we present treewidth-based algorithms for Roman Domination, Weak Roman Domination and Secure Domination. These are all problems that have been studied extensively. For Roman Domination we present an $\mathcal{O}^*(3^t)$ time algorithm for graphs of treewidth t. This is an improvement on the current literature. For Weak Roman Domination and Secure Domination we present an $\mathcal{O}^*(9^t)$ time and $\mathcal{O}^*(8^t)$ time algorithm respectively for graphs of treewidth t. To our knowledge these results are the first treewidth-based algorithms for these problems.

1 Introduction

We will construct exact exponential-time and treewidth-based algorithms for a series of defence-like domination problems. Defence-like domination problems are variants of Dominating Set in which we try to defend a graph from a sequence of attacks. These problems originate from Roman Domination, where the goal is to defend the roman empire using as few legions as possible. In this thesis we will introduce the notion of defence-like domination problems and extensively study some of them, looking into exact exponential-time and treewidth-based algorithms. The definitions of defence-like domination problems are generalisations of Weak Roman Domination, introduced by Henning et al. [10]. This problem in turn originates from Roman Domination, introduced by Cockayne et al. [6], where Roman Domination is a variant of Dominating Set.

For Dominating Set the currently best known exact exponential-time algorithms are the results from Iwata [11]. These results are an $\mathcal{O}^*(1.4864^n)$ time algorithm using polynomial space and an $\mathcal{O}^*(1.4689^n)$ time algorithm using exponential space. There also exists a treewidth algorithm for Dominating Set running in $\mathcal{O}^*(3^t)$ time (for graphs of treewidth t), by van Rooij et al. [21].

Roman Domination is a variant of Dominating Set, where the rules are inspired by a decree from emperor Constantine of the roman empire. These rules state that a vertex can protect itself when it has 1 legion present and it can protect itself and all its neighbours when it has 2 legions present. This is because Constantine required any city to have 2 legions present in order to allow one of these legions to move to a nearby city. For Roman Domination there also exist exact exponential-time algorithms. The best known exact exponential-time algorithms are an $\mathcal{O}^*(1.5673^n)$ time algorithm using polynomial space and an $\mathcal{O}^*(1.5014^n)$ time algorithm using exponential space, by Shi et al. [17]. These results are actually just applying the results from Nederlof et al. [14] to this problem. An algorithm similar to the treewidth algorithm for Dominating Set can also be applied to Roman Domination with some adjustments, yielding an $\mathcal{O}^*(3^t)$ time algorithm. This algorithm is presented in this thesis. This is an improvement on the results

from Fernau [8] and Peng et al. [15]. These are 2 independent results that both yielded an $\mathcal{O}^*(5^t)$ time treewidth algorithm for Roman Domination.

Weak Roman Domination is inspired by Roman Domination, but it relaxes the strict rules of the decree from Constantine. This is done in the following way. In Weak Roman Domination we allow legions to move to neighbouring vertices when this does not result in an undefended vertex, where an undefended vertex is defined as a vertex for which the closed neighbourhood does not contain any vertex with 1 or more legions. This means that a vertex that only has a single legion on it can protect neighbouring vertices, but only if it can safely move to these neighbours. For Weak Roman Domination there exists an $\mathcal{O}^*(2^n)$ time algorithm using exponential space and an $\mathcal{O}^*(2.2279^n)$ time algorithm using polynomial space by Chapelle et al. [5], however to our knowledge no treewidth algorithm exists yet. We note that even though Roman Domination and Weak Roman Domination use legions, we will be using guards. This is because the term legion comes from the roman background of this problem and we will be generalising this. This means that for readability we will always refer to these legions as guards even in the context of Roman Domination and Weak Roman Domination.

Another problem that relates closely to Weak Roman Domination is Secure Domination, introduced by Cockayne et al. [7]. Secure Domination is essentially Weak Roman Domination, with the exception that we cannot place 2 guards on one vertex, but at most 1. For this problem there exists a branching algorithm by Burger et al. [4] for which the run time is $\mathcal{O}^*(2^{n-s-r})$, where s and r are the number of support vertices and redundant vertices of the graph respectively, as well as a binary programming algorithm by Burger [3]. This last result has recently been improved by Burdett et al. [1].

2 Preliminary Definitions

In this thesis we use a number of definitions from [5]. These definitions greatly improve readability and therefore we introduce them here. Let $G = (V, E)$ a graph and $f : V \to \mathbb{N}$ a function. We will call such a function a *guard assignment*. We say that a vertex $v \in V$ is *secured* if $f(v) \geq 1$, and *unsecured* otherwise. Similarly, a vertex $v \in V$ is said to be *defended* if there exists a $u \in N[v]$ such that $f(u) \geq 1$. Otherwise, v is said to be *undefended*. We note that we use $N[v]$ to refer to the closed neighbourhood of v and $N(v)$ to refer to the open neighbourhood of v (i.e. $N[v] = N(v) \cup \{v\}$). We use the function $f_{u\to v}$ to refer to the function that is created by moving a guard from u to v and this function is defined as follows.

$$f_{u\to v}(w) = \begin{cases} f(w) + 1 & \text{if } w = v \\ f(w) - 1 & \text{if } w = u \\ f(w) & \text{otherwise} \end{cases}$$

Definition 2.1. (Safely-Defended Vertex). Let $v \in V$ be any vertex and f be a guard assignment. We say that v is *safely defended* under f if one of the following holds:

- v is secured, i.e. $f(v) \geq 1$;
- or, $\exists u \in N(v) : f(u) \geq 2$;
- or, $\exists u \in N(v) : f(u) = 1$ and the vertices undefended under $f_{u\to v}$ are the same as the ones undefended under f, i.e., $f_{u\to v}$ creates no new undefended vertex.

If v is unsecured, but safely defended (i.e. one of the last 2 conditions hold), then we say that the vertex u *safely defends* v, or v is *safely defended by* u, or u is the *safe defender* of v.

If v is defended but not safely defended, we say that v is *non-safely defended*. We will refer to these definitions often in the rest of this thesis.

For completeness sake we will also include the notion of weakly defended vertices. This definition is less important than the other definitions, but it is used in the literature. When a vertex v is non-safely defended, we know that v is unsecured and for every secured neighbour u of v, $f(u) = 1$ and $f_{u\to v}$ contains at least one undefended vertex $w \in N(u)$. We will refer to such a vertex w as being *weakly defended by* u or simply *weakly defended*. Note that a vertex that is weakly defended has exactly one secured neighbour.

2.1 Problem Definitions

We will now give the complete problem definitions for the graph problems discussed in Section 1. These are all known problems that are discussed in this thesis. We start with the definition of Dominating Set.

Dominating Set:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum size of a dominating set, denoted by $\gamma(G)$.

A dominating set is a set $D \subseteq V$ such that for every vertex $v \in V$ there exists a vertex $u \in D$, such that $v \in N[u]$.

Next we give the definition of Roman Domination. In the following we define the cost κ of any function $f : V \to \mathbb{N}$ by $\kappa = \sum_{v \in V} f(v)$.

Roman Domination:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum cost of a roman domination function f, denoted by $\gamma_r(G)$.

A roman domination function (rd-function for short) is a function $f : V \to \{0, 1, 2\}$ for which every vertex $v \in V$ is either secured (i.e. $f(v) \geq 1$) or there exists a $u \in N(v)$, such that $f(u) = 2$.

Next we can define Weak Roman Domination and Secure Domination. We follow the definition of Weak Roman Domination from Chapelle et al. [5].

Weak Roman Domination:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum cost of a weak roman domination function f, denoted by $\gamma_w(G)$.

A weak roman domination function (wrd-function for short) is a function $f : V \to \{0, 1, 2\}$ for which every vertex $v \in V$ is either secured or there exists a $u \in N(v)$, such that $f(u) \geq 1$ and in $f_{u \to v}$ every vertex is defended. Notice that a guard assignment f is a wrd-function if and only if every vertex $v \in V$ is safely defended under f and thus there are no weakly defended vertices.

Secure Domination:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum cost of a secure domination function f, denoted by $\gamma_s(G)$.

A secure domination function (sd-function for short) is a function $f : V \to \{0, 1\}$ for which the same conditions must hold as for a wrd-function. The only difference between an sd-function and a wrd-function is that in an sd-function a vertex can have at most one guard. This means that again every vertex $v \in V$ is either secured or there exists a $u \in N(v)$, such that $f(u) \geq 1$ and in $f_{u \to v}$ every vertex is defended.

3 Defence-Like Domination Problems

We will now give our definition of defence-like domination problems. To achieve this definition we will generalise the definition of Weak Roman Domination in every reasonable way (e.g. the number of guards allowed on a vertex, or the number of guards that are allowed to move during an attack). The resulting definition will be our definition of defence-like domination problems. This general definition will encapsulate several known variants. In the rest of this section we will first give our general definition defence-like domination problems (Section 3.1). We will follow this by more concrete definitions of the possible variants in Section 3.2. First we will look at variants with different defence strategies in Section 3.2.1. All of these strategies have been defined in earlier works, but to our knowledge no attempts at either exact exponential-time

or treewidth algorithms have been made for these variants. Next we will introduce a number of parameters, which will lead us to our parameterized problem variants (Section 3.2.2). To our knowledge the only instances of our parameterized problem variants that have been studied are Weak Roman Domination and Secure Domination, as well as a-Attack Defensive Domination (under the name k-Weak Roman Domination) by Burger et al. [2]. Furthermore a variant in which the number of turns is infinite has been studied more widely as Eternal Domination [9]. However to our knowledge this problem is not in NP, since checking a solution means checking an infinite number of possible moves. Therefore we have chosen to not include this problem into our parameterized variants.

3.1 Definition of Defence-Like Domination

In this section we will give our general definition of defence-like domination problems. In Section 3.2.2 we will use this definition to define a number of more concrete parameterized variants, which will be the variants we will study in this thesis. In order to give this general definition consider the following general setting.

For a graph $G = (V, E)$, a guard assignment or guard function is a function $f : V \to \{0, 1, ..., c\}$ where, for every $v \in V$, the value of $f(v)$ is defined as the number of guards on vertex v. As stated before the guards serve the same purpose as the legions in Weak Roman Domination, but with a much less Roman name. The capacity c is the maximum number of guards that can be assigned to a single vertex. An attack sequence is defined as $A = \{A_0, A_1, ..., A_k\}$, where $A_i \subseteq V$ for all $0 \leq i \leq k$. Here A_i represents the set of vertices that are under attack at turn i. We will define the maximum attack size of sequence A by $a = \max_{0 \leq i \leq k}\{|A_i|\}$.

A step is defined by an edge $e = \{u, v\} \in E$ and represents a single guard g moving from u to v. We say that guard g has moved one step along edge e. This results in a new guard assignment $f_{u \to v}$ as defined before. During an attack we can move, for a given size h and number of steps s, a subset of our guards of size at most h, at most s steps each in order to defend our graph G. A collection of moves that satisfies these two restrictions is called a valid collection of moves.

We say that f can defend A_0, if there exists a valid collection of moves that generates a new assignment f_0, from f, such that $\forall u \in A_0 : f_0(u) \geq 1$. Inductively we say that f can defend A_{i+1} if f can defend A_i and there exists a valid collection of moves generating f_{i+1}, from f_i, such that $\forall u \in A_{i+1} : f_{i+1}(u) \geq 1$. We call f a defence-like domination function (dld-function for short), if f can defend every attack in the sequence A, for every possible sequence A.

The cost of a dld-function is defined as $\kappa = \sum_{v \in V} f(v)$. The cost represents the number of guards needed for the assignment. Defence-like domination problems are then defined as follows. We have as input a graph $G = (V, E)$ and as output we require a dld-function f of minimum cost.

3.2 Variants

In this thesis we will present a number of variants of defence-like domination problems. In particular we will present some variants with different defence strategies (Section 3.2.1) as well as introduce a couple of parameterized variants (Section 3.2.2).

3.2.1 Strategies

In our general definition of defence-like domination problems (Section 3.1), we talk about moving guards to defend vertices from attack sequences. However, which guards to move and which vertices will be attacked, after we have chosen our guard assignment, is not yet discussed. In this section we will present 3 different problem variants, that handle this. We say that we have an attacking strategist. He or she constructs the attack sequence. We also have a defending strategist. He or she decides at every attack which guards to move in order to defend this attack. We will always assume the defending strategists to be perfect, i.e. he or she will always make the best choice based on the information available. According to [13] there are two problem variants regarding the way the attacking strategist chooses and reveals the attack sequence for defence-like domination problems.

1. **Offline:** In the offline version the entire attack sequence is chosen and revealed beforehand. The defending strategist can use its knowledge of the entire sequence to find a sequence of moves to defend against this.

2. **Online:** In the online version the attack sequence is chosen but not revealed beforehand and the defending strategist has to move the guards based only on its knowledge of the current attack.

There is also a third option in which the attacking strategist does not choose the attack sequence beforehand but chooses its next attack based on the result of the previous attacks. In our case this is exactly the same as the online variant, since our assignment of guards has to be able to defend every possible attack sequence as also stated in [13]. This gives us two different strategic variants for defence-like domination problems.

So far we have been assuming that we have a perfect defending strategist that can make the best decisions in the defence against attacks. If we drop this requirement we get the third and last strategic variant, described in [2].

3. **Foolproof:** In the foolproof variant we require the assignment of guards to be such that at every attack any successful defensive move will result in a successful defence of the entire sequence. In other words it doesn't matter which defensive choices we make at every attack, because each defensive choice is just as effective.

Unless explicitly stated the reader may always assume we are referring to the online variant of a problem.

3.2.2 Parameters

In Section 3.1 we gave a general definition for defence-like domination problems. In doing so we defined a number of parameters. These are the following.

- The number of turns, k.
- The maximum attack size, a.
- The number of guards that are allowed to move each turn, h.
- The maximum number of guards per vertex, c.
- The maximum number of steps per guard, s.

Using these parameters we can define the variants we will study. We will always assume the capacity c to be infinite and the rest of these parameters to be 1 unless stated otherwise. e.g. k-Turn Defensive Domination would be a variant in which c is infinite, so $f : V \to \mathbb{N}$, k is the parameter that can vary and a, t and s are all equal to 1. This results in the following list of parameterized problem variants.

- k-Turn Defensive Domination
- a-Attack Defensive Domination
- t-Move Defensive Domination
- c-Capacitated Defensive Domination
- s-Step Defensive Domination

Using these definitions we see that we can refer to Weak Roman Domination as simply Defensive Domination. This is because even though Defensive Domination would allow any number of guards on a vertex, having more than 2 guards would not improve the assignment. Following the same logic Dominating Set will be 0-Turn Defensive Domination and Secure Domination is 1-Capacitated Defensive Domination. Roman Domination is not part of this definition, because with Roman Domination a vertex that has only one guard is not allowed to defend a neighbouring vertex. Furthermore Eternal Domination could be referred to as ∞-Turn Defensive Domination. We note that we can also use this definition to create a problem variant with more than one parameter, e.g. 2-Turn 1-Capacitated Defensive Domination.

4 k-Turn Defensive Domination

In the rest of this thesis we will focus on the k-Turn Defensive Domination problem. We will give an exact exponential-time algorithm for k-Turn Defensive Domination, that runs in $\mathcal{O}^*(4^n)$ time, in Section 4.3. After this we give an $\mathcal{O}^*(3^t)$ time treewidth algorithm for Roman Domination, for graphs of treewidth t, in Section 6. This is followed by an $\mathcal{O}^*(9^t)$ time treewidth algorithm for Weak Roman Domination (or Defensive Domination), for graphs of treewidth t, in Section 7. This algorithm can easily be transformed into an $\mathcal{O}^*(8^t)$ time treewidth algorithm for Secure Domination (or 1-Capacitated Defensive Domination).

4.1 Problem Definitions

Since we will focus on k-Turn Defensive Domination and k-Turn 1-Capacitated Defensive Domination in the rest of this thesis, we will first give a complete problem definition for both these problems. These definitions are the same as the definition in 3.1, but stripped down to only include the parameter k.

k-Turn Defensive Domination:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum cost of a k-turn defensive domination function f, denoted by $\gamma_k(G)$.

A k-turn defensive domination function (k-tdd-function) can again be defined inductively in the following way. A 1-tdd-function is the same as a wrd-function. A k-tdd-function is then a function $f : V \to \mathbb{N}$ for which every vertex $v \in V$ is defended and for every vertex $v \in V$ for which $f(v) = 0$ there exists a $u \in N(v)$, such that $f(u) \geq 1$ and $f_{u \to v}$ is a $(k-1)$-turn defensive domination function. This inductive definition guarantees that f is a k-tdd-function if and only if f can defend against any attack sequence $A = \{a_0, a_1, ..., a_k\}$, where $a_i \in V$ is a single vertex.

k-Turn 1-Capacitated Defensive Domination:	
Input:	An undirected graph $G = (V, E)$.
Output:	The minimum cost of a k-turn 1-capacitated defensive domination function f, denoted by $\gamma_{kc}(G)$.

A k-turn 1-capacitated defensive domination function (k-tcdd-function) can again be defined inductively in the following way. A 1-tcdd-function is the same as a sd-function. A k-tcdd-function is then a function $f : V \to \{0, 1\}$ for which every vertex $v \in V$ is defended and for every vertex $v \in V$ for which $f(v) = 0$ there exists a $u \in N(v)$, such that $f(u) \geq 1$ and $f_{u \to v}$ is a $(k-1)$-turn 1-capacitated defensive domination function. This inductive definition guarantees that f is a k-tcdd-function if and only if f can defend against any attack sequence $A = \{a_0, a_1, ..., a_k\}$, where $a_i \in V$ is a single vertex.

4.2 NP-Completeness

Roman Domination is known to be NP-complete. This is stated in [6], with a reference to private communication from McRae. Weak Roman Domination is also known to be NP-complete, as proven by Henning et al. [10]. The same proof also holds for Secure Domination. For our k-turn variants we will give a modified version of this proof to show that these are also NP-complete for fixed k. Before we give this proof we first need the following lemma.

Lemma 4.1. *Let $P = \{v_1, v_2, ..., v_{2k+2}\}$ be a path on $2k+2$ vertices. Let us use this path as an input graph to either k-Turn Defensive Domination or k-Turn 1-Capacitated Defensive Domination. Let $Q = \{\{v_1, v_2\}, \{v_3, v_4\}, ...\{v_{2k+1}, v_{2k+2}\}\}$, be a partitioning of P in pairs. For any optimal k-tdd-function or k-tcdd-function f, the following holds: $\forall q \in Q : \sum_{v \in q} f(v) = 1$. In other words, any optimal solution will always have exactly one guard on one of the 2 vertices of each element from Q. We will call such a function f a pairwise guard assignment.*

Proof. We first note that any pairwise guard assignment f is always both a k-tdd-function and a k-tcdd-function. This is because for every such f every vertex is defended. Either it is secured or it has a secured neighbour, namely the other half of the corresponding pair in Q. We can use this assignment to defend against any number of attacks since moving a guard from one vertex in an element of Q to the other vertex in this element, will again result in a pairwise guard assignment, which means that every vertex is still defended. Any such assignment is also optimal, because of the following reasons. Firstly placing more than one guard on a single vertex could only make its neighbouring vertices more safely defended. Since there are 2 such neighbours, we would use at least 2 guards to defend 3 vertices, while now we only use 1 guard per 2 vertices (note that our path has an even amount of vertices). This means that we can only get an optimal solution by placing at most one guard on each vertex. Secondly placing guards on fewer vertices would always result in at least two guards g and g' having to guard 3 vertices, in stead of 2 vertices per guard. This means that using $k-1$ attacks we could force every other guard to move to exactly the position we want it, leaving the last move to force g or g' to move to one of its neighbours leaving the other undefended. □

Theorem 4.2. k-Turn Defensive Domination *and* k-Turn 1-Capacitated Defensive Domination *are* NP*-complete for every fixed* k*.*

Proof. Let $H = (V_H, E_H)$ be a graph. For every fixed k it is easy to see that both k-Turn Defensive Domination and k-Turn 1-Capacitated Defensive Domination are in NP. This is because for k-Turn Defensive Domination any optimal solution would never have more than k guards on a single vertex. Furthermore we can guess in polynomial time at either a function $f : V_H \to \{0, 1, ..., k\}$ or a function $f : V_H \to \{0, 1\}$ and verify whether this is a solution. This can be done by simply checking all possible attack sequences and all possible defensive moves against these. This approach would take $\mathcal{O}(n^k)$ time. Next we give a polynomial time reduction from Dominating Set to both k-Turn Defensive Domination and k-Turn 1-Capacitated Defensive Domination.

Let $G = (V_G, E_G)$ be the input graph for Dominating Set. We construct a graph H by adding a path of length $2k + 2$ to every vertex of G. This can again be done in polynomial time. We will now show that $\gamma_k(H) = \gamma_{kc}(H) = \gamma(G) + (k + 1)|V_G|$, which concludes our proof.

Let $f : V_H \to \{0, 1, ..., k\}$ be a k-tdd-function (or a k-tcdd-function) of minimum cost, $v \in V_G \subset V_H$ a vertex and $P_v = \{v, v_1, v_2, ..., v_{2k+2}\}$ the path of length $2k + 2$ added to v. We know from Lemma 4.1 that any optimal solution on a path of length $2k + 2$ is always a pairwise guard assignment. The path $P_v \setminus \{v\}$ is only connected to the rest of the graph through vertex v. This means that the vertices in $P_v \setminus \{v\}$ need at least $k + 1$ vertices in any k-tdd-function or k-tcdd-function.

Now suppose that $\sum_{u \in P_v} f(u) = k + 1$. This means that we know that the guard assignment in $P_v \setminus \{v\}$ is a pairwise guard assignment and $f(v) = 0$. This also means that if the first attack in the sequence is at vertex v_2, the guard that is at either v_1 or v_2 will have to be at v_2 after this attack. This would mean that v would be undefended after this attack unless there is a secured vertex w in the neighbourhood of v.

On the other hand suppose that $\sum_{u \in P_v} f(u) \geq k + 2$, then we may assume that $f(v) \geq 1$ and we still have a pairwise guard assignment for the vertices in $P_v \setminus \{v\}$. This is because any extra guard placed on the vertices in $P_v \setminus \{v\}$, would be more effective on vertex v. This means that the set of vertices defined by $S = \{u \in V_G \mid f(u) \geq 1\}$ (the set of secured vertices) is a dominating set of G. Therefore $\gamma(G) \leq |S|$. Furthermore if $v \in S$ then $\sum_{u \in P_v} f(u) \geq k + 2$, while if $v \notin S$ then $\sum_{u \in P_v} f(u) = k + 1$. This means that $\gamma_k(H) \geq (k + 2)|S| + (k + 1)(|V_G| - |S|) = |S| + (k + 1)|V_G| \geq \gamma(G) + (k + 1)|V_G|$. The exact same equation holds for $\gamma_{kc}(H)$.

On the other hand, let D be a minimum dominating set. Let $f' : V_H \to \{0, 1\}$ be the function defined as follows. For every vertex $v \in V_G$ we place one guard on every odd vertex $(v_1, v_3, ..., v_{2k+1})$ for every path P_v. Also if $v \in D$ we also place one guard on v and otherwise we place no guard on v. This function f' is both a k-tdd-function and a k-tcdd-function, since any unsecured vertex in V_G is defended by a neighbour in V_G and after the first attack every vertex $v \in V_G$ is defended by its path P_v for the rest of the attacks. Also any vertex not in V_G is defended for any number of attacks as shown in Lemma 4.1. This means that $\gamma_k(H) \leq |D| + (k + 1)V_G = \gamma(G) + (k + 1)V_G$. The same again holds for $\gamma_{kc}(H)$. This means that $\gamma_k(H) = \gamma_{kc}(H) = \gamma(G) + (k + 1)|V_G|$, which is what we wanted to prove and thus this completes our proof. □

4.3 Exact Exponential-Time Algorithm

In this section we will present an $\mathcal{O}^*(4^n)$ time algorithm for k-TURN DEFENSIVE DOMINATION. This result is based on the following observation.

Lemma 4.3. *Any instance of any defence-like domination problem as defined in this thesis has a solution of cost at most n.*

Proof. This can be done by simply placing a guard at every vertex in the graph $G = (V, E)$. This gives us the guard assignment f, defined by $\forall v \in V : f(v) = 1$. The cost of this guard assignment is n, since $\kappa = \sum_{v \in V} f(v) = \sum_{v \in V} 1 = |V| = n$. At every possible attack we can always choose to not move any guards. This will always result, at turn i, in an assignment f_i where $f_i(v) = 1$ for each $v \in V$, since this is the assignment we started with. This also means that $f_i(u) \geq 1$ for all $u \in A_i$ for all $i \leq k$. Therefore, according to our definition of defence-like domination problems, this is a solution to defence-like domination problem. □

This result tells us that we will never have to consider solutions that have a cost higher than n, since in those cases we already have a better solution. For our algorithm we will be enumerating every possible solution of cost at most n. We will do so using compositions.

Definition 4.4. (Composition). A composition of an integer n is an ordered sequence of strictly positive integers that sum to n. An l-composition is a composition of length exactly l.

It is a standard result in combinatorics that for any pair of integers n and l there are $\binom{n-1}{l-1}$ different l-compositions of n [18]. This leads us to the following result.

Lemma 4.5. *There are $\binom{2n}{n}$ different possible guard assignments of cost at most n.*

Proof. We will index the vertices of our graph as $V = \{v_1, v_2, ..., v_n\}$. Next we will consider every possible $(n+1)$-composition of $2n+1$. For every possible composition $C = \{c_0, c_1, ..., c_n\}$ we can construct a corresponding guard assignment. We say that $f(v_i) = c_i - 1$. We know that $c_i \geq 1$ for all $0 \leq i \leq n$, since they are strictly positive integers. Also, since we consider every possible way to sum $n+1$ such integers to $2n+1$, we know that $\sum_{i=0}^{n}(c_i - 1) = n$. In other words every c_i is at least 1 and we consider every possible way to distribute the remaining n over these $n+1$ integers. We use c_0 to indicate that we don't use these guards, thus this technique results in every possible guard assignment using at most n guards. Since there are $\binom{2n}{n}$ different $(n+1)$-composition of $2n+1$, there are also $\binom{2n}{n}$ different possible guard assignments of cost at most n. □

Finally we can combine these 2 results into the following algorithm.

Theorem 4.6. *There exists an $\mathcal{O}^*(4^n)$ time algorithm, using polynomial space, for every variant of* DEFENSIVE DOMINATION*, for which we can check a solution in polynomial time.*

Proof. Because of Lemma 4.3, we know that we only need to consider assignments of cost at most n. Also, because of Lemma 4.5 we know that there are only $\binom{2n}{n}$ possible assignments of cost at most n. So we can enumerate all these possible assignments in $\mathcal{O}^*(\binom{2n}{n})$ time. We know that $\binom{x}{x/2} \leq 2^x$, since $\binom{x}{x/2}$ is equal to the number of subsets of size $x/2$ of a collection of size x and the total number of subsets of a collection of size m is equal to 2^m. Therefore $\binom{2n}{n} \leq 2^{2n} = 4^n$. This means that we can enumerate every possible assignment in $\mathcal{O}^*(4^n)$ time. Since we can check each assignment in polynomial time, we can find the optimal solution in $\mathcal{O}^*(4^n)$ time. This approach uses polynomial space, since we do not store anything other than the current optimal solution. □

For k-TURN DEFENSIVE DOMINATION checking a solution will take $\mathcal{O}(n^k)$ time. We can improve on this in the following way.

Theorem 4.7. *There exists an $\mathcal{O}(4^n mk)$ time algorithm, using $\mathcal{O}(4^n)$ space, for k-*TURN DEFENSIVE DOMINATION.

Proof. For this algorithm we will create a table, A_0, which we will index with every possible assignment of cost at most n. Following the same logic as in Theorem 4.6 we know that we can achieve this in $\mathcal{O}(4^n)$ time and that this table will take up $\mathcal{O}(4^n)$ space. For each of these assignments we store a Boolean value indicating whether they are a 0-tdd-function or not (i.e. we set $A_0(f) = True$ if f is a 0-tdd-function and $A_0(f) = False$ otherwise). Then we enumerate for $i = 1$ until k:

Create a new table A_i and drop table A_{i-2} if it exists. Enumerate every possible assignment f of cost at most n, check for every vertex v if it is secured or if we can move a guard from a neighbour, u, to this vertex, such that $A_{i-1}(f_{u\to v}) = True$. If this is the case we set $A_i(f) = True$, else we set $A_i(f) = False$.

For any assignment f, $A_k(f)$ will always indicate whether f is an k-tdd-function. We will prove this with induction. First we know that we initialise A_0 such that $A_0(f)$ will always indicate whether f is an 0-tdd-function. Next we assume that for every assignment f, $A_i(f)$ will always indicate whether f is an i-tdd-function. Following the algorithm we can see that we will set $A_{i+1}(f) = True$ if and only if f is an $(i+1)$-tdd-function. This is because we only set this entry to $True$ if and only if there exists, for every possible attack, a move from an i-tdd-function that defends against this. This means that for any assignment f, $A_k(f)$ will always indicate whether f is an k-tdd-function. This means that after running this algorithm we can simply enumerate A_k to find the optimal solution.

This algorithm runs in $\mathcal{O}(4^n mk)$ time, because our tables have size $\mathcal{O}(4^n mk)$ and we create k copies of this table. When filling these copies we first check if each entry is a 0-tdd-function. We can check this in $\mathcal{O}(m)$ time, because we have to check for each vertex all of its neighbours. This means that we look at each edge at most twice. Then, for each step in the algorithm, we again have to check for each vertex all of its neighbours. This again takes $\mathcal{O}(m)$ time. Since at step i we only need to store A_i and A_{i-1} we know that this algorithm uses $\mathcal{O}(4^n)$ space. □

Since k-Turn 1-Capacitated Defensive Domination has only $\mathcal{O}(2^n)$ possible assignments we also know the following.

Corollary 4.7.1. *There exists an $\mathcal{O}(2^n mk)$ time algorithm, using $\mathcal{O}(2^n)$ space, for k-Turn 1-Capacitated Defensive Domination.*

5 Treewidth Algorithms

The rest of this thesis will be focused on treewidth algorithms for instances of k-Turn Defensive Domination. We will begin by defining treewidth and tree decompositions in Section 5.1 and follow this by an explanation of how we can use this to construct a dynamic programming algorithm in Section 5.2.

5.1 Treewidth and Tree Decompositions

Treewidth and tree decompositions were introduced by Robertson and Seymour [16] and are defined as follows.

Definition 5.1. (Tree Decomposition). Given a graph $G = (V, E)$, a *tree decomposition* of G is a tree T, in which each node $x \in T$ has an associated set of vertices $X_x \subseteq V$ (called a *bag*) such that $\bigcup_{x\in T} X_x = V$ and the following properties hold.

1. For every edge $(u, v) \in E$, there exists a bag X_x, such that $(u, v) \in X_x$.
2. For every vertex $v \in V$, the bags containing v form a connected subtree, i.e., if $v \in X_x$ and $v \in X_y$, then $v \in X_z$ for all nodes z on the path from node x to node y in T.

The *width* of a tree decomposition T is defined as $\max_{x\in T}\{|X_x|\} - 1$. The *treewidth* of a graph G is the minimum width over all possible tree decompositions of G.

In order to construct a dynamic programming algorithm over a tree decomposition it is usually the best option to do this on nice tree decompositions, which were introduced by Kloks [12]. The definition presented below is from van Rooij [21] and differs slightly from the definition by Kloks.

Definition 5.2. (Nice Tree Decomposition). A *nice tree decomposition* is a tree decomposition T that has an assigned root node $\rho \in T$, with $X_\rho = \emptyset$ and every node $x \in T$ is of one of the following types.

Leaf node x is a leaf of T with $X_x = \emptyset$.

Introduce node x is an internal node of T with one child node y and $X_x = X_y \cup \{v\}$, for some $v \notin X_y$. The node is said to *introduce* the vertex v.

Forget node x is an internal node of T with one child node y and $X_x = X_y \setminus \{v\}$, for some $v \in X_y$. The node is said to *forget* the vertex v.

Join node x is an internal node of T with two child nodes l and r and $X_x = X_l = X_r$.

The reason why nice tree decompositions are often used in algorithms is because they add a structure to tree decompositions, which is very useful for constructing dynamic programming algorithm. Also given a tree decomposition of $\mathcal{O}(n)$ nodes, we can find a nice tree decomposition of equal width and $\mathcal{O}(n)$ nodes in $\mathcal{O}(n)$ time [12].

5.2 Dynamic Programming on Tree Decompositions

We will explain how we can use these definitions to construct a dynamic programming algorithm on tree decompositions. For a more extensive and mathematical introduction see for example van Rooij [20]. In this thesis we will focus on using this approach to solve the counting variant of computational problems. In Section 2.1 and Section 4.1 we gave the minimisation variant of the relevant problems. In the counting variant the goal is no longer to find the cost of the minimum solution. The goal is to find, for each value $0 \leq \kappa \leq N$, the number of solutions of cost exactly κ, where N is the largest viable cost of any solution. It is easy to see that by solving this problem we also solve the minimisation problem as we can simply loop through each κ until we find the first non zero entry.

Given a nice tree decomposition T, we define for each node $x \in T$: T_x as the subtree of T rooted at x and $G_x = (V_x, E_x)$ as the subgraph of G, where $V_x = \bigcup_{y \in T_x} X_y$ and E_x is the subset of edges from E that have both endpoints in V_x, i.e. $E_x = \{(u, v) \in E \mid u \in V_x \wedge v \in V_x\}$. We note that following these definitions we can see that $T = T_r$ and $G = G_r$. Using the definition of a tree decomposition we can also see that X_x is a *separator* separating $V_x \setminus X_x$ from $V \setminus V_x$ in G, i.e., there are no edges in E that have an endpoint in $V_x \setminus X_x$ and an endpoint in $V \setminus V_x$. Any graph we can obtain by adding any number of vertices and edges to G_x, with the restriction that any edges we add can only have at most one endpoint in the bag X_x and no endpoints in $V_x \setminus X_x$ is called an *extension* of G_x. In particular G is an extension of G_x.

In our treewidth-based algorithms we will assume that we have a graph problem that we are trying to solve $\mathcal{P}$ on a graph G and are given a nice tree decomposition T of G. We assume that a solution of $\mathcal{P}$ is a function f that maps V to any specific domain. We will construct a dynamic programming algorithm on T in a bottom up fashion. To do so, we will need to define a *partial solution* of $\mathcal{P}$ on G_x, where a partial solution on G_x is any function f, defined on the domain V_x, for which there exists an extension of G_x that turns f into a valid solution of $\mathcal{P}$. If y is a child node of x in T, then G_x is an extension of G_y. This means that if we apply a partial solution for G_x on G_y we still have a partial solution. This means that if we know every partial solution for every child node of x, we can use this to compute every partial solution for node x.

We don't need to store every partial solution for every node, since the only vertices in node x for which the neighbourhood can increase in an extension, are the vertices in the bag X_x. We will therefore define a *colouring* of a bag X_x as a function $c : X_x \to S$, where S is a set of *states* that, for every vertex $v \in X_x$, encodes information about v and its neighbourhood (and every possible neighbourhood under an extension). S is also called a *state set*, and is specific to the problem $\mathcal{P}$. If we can effectively encode the necessary information in our state set, we do not need to store every possible partial solution for any node x. It will then suffice to only store for every possible colouring c of X_x, the number of partial solutions of each cost $0 \leq \kappa \leq N$, where N is the largest viable cost of a solution to $\mathcal{P}$ on G. We will call the table in which we store this information for each node a *memoisation table*. Note that using this technique we can compute this memoisation table for every node of T in a bottom up fashion. The size of the memoisation table will be $\mathcal{O}^*(|S|^{|X_x|})$, which means that we expect such an algorithm to run in $\mathcal{O}^*(|S|^t)$ time, where t is the width of T. Also note that, since the bag X_r is empty and $G = G_r$, every partial solution on G_r is a solution on G. Therefore the memoisation table of r stores the number of solutions of $\mathcal{P}$ on G for each cost $0 \leq \kappa \leq N$, so an algorithm applying this technique will solve the counting problem of $\mathcal{P}$.

6 Treewidth Algorithm for Roman Domination

In order to best illustrate the concepts from Section 5.2, we will give an $\mathcal{O}^*(3^t)$ time algorithm for Roman Domination. To our knowledge this is a new result, but the algorithm is very similar to the algorithm for Dominating Set, by van Rooij et al. [21]. This algorithm can almost directly be applied to this problem yielding an $\mathcal{O}^*(4^t)$ time algorithm, but with an additional observation we get an $\mathcal{O}^*(3^t)$ time algorithm. We note that Roman Domination is not included in our definition of defence-like domination problems. However the algorithm is relatively easy to follow, so it is an ideal start to get familiarised with treewidth algorithms for defence-like domination problems. In the rest of this section we present a treewidth algorithm that solves the counting variant of Roman Domination, as explained in Section 5.2. This means that the problem we are trying to solve is the following. Given a (nice) tree decomposition of graph $G = (V, E)$ of width t, find the number of rd-functions of each cost $0 \leq \kappa \leq n$. Recall that an rd-function is a function $f : V \to \{0, 1, 2\}$, such that for every $v \in V$ either $f(v) \geq 1$ or $\exists u \in N(v) : f(u) = 2$. Also note that just as with any instance of defence-like domination, simply putting a guard on every vertex is a solution, so we do not need to consider solutions of size larger than n.

6.1 States

As discussed in Section 5.2 we will need to define a state set S for this problem. This state set will allow us to index the memoisation tables A_x for every $x \in T$. These are tables that store, for every colouring $c : X_x \to S$ and each cost κ, the number of partial solutions of cost exactly κ that satisfy the colouring c. We will slightly abuse notation when referring to colourings. We will use a colouring c to refer to both the function that maps X_x to S, as well as the set of states that X_x is mapped to. This allows us to use the notation $c \times \{s\}$ to indicate that we extend the colouring with a vertex v, for which $c(v) = s$. Any entry in A_x is denoted by $A_x(c, \kappa)$, which represents the number of partial solutions of cost exactly κ that satisfy the colouring c.

The state set that we will use for this algorithm is the following.

$$S = \{2, 0_0, 0_?\}$$

In this state set we first have the state 2. This state is very straightforward. For a vertex v that has state 2, we know $f(v) = 2$. If a vertex v has state 0_0 or $0_?$, this means that $f(v) = 0$ (or $f(v) = 1$ as we will explain later). The subscript is used to store extra information about these vertices. A vertex of state 0_0 has no neighbours for which $f(v) = 2$. A vertex of state $0_?$ may have a neighbour for which $f(v) = 2$, but it is not required. This means that, looking at the definition of a rd-function, a vertex of state 0_0 is not allowed in a solution, but a vertex of state $0_?$ might be. We note that the states 0_0 and $0_?$, might not be the most straightforward choice. This choice is an optimisation and it suffices to count solutions. We also note that we do not have a state to explicitly indicate that $f(v) = 1$. This is another optimisation. Because we know that any vertex that does not have a neighbour with 2 guards must have 1 guard itself, we can simply count a vertex of state 0_0 as a vertex with one guard when we forget this vertex.

6.2 Algorithm

We will show, for each type of node, that we can compute the table A_x.

LEAF NODE:

Let x be a leaf node in T. In the leaf nodes we only have empty bags and therefore the only possible colouring is $\emptyset$ with cost 0.

$$A_x(\emptyset, \kappa) = \begin{cases} 1 & \text{if } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

INTRODUCE NODE:

Let x be an introduce node in T with child node y that introduces the vertex v, and $c \in S^{X_y}$. In an introduce node we will construct the result table A_x. We can do this in the following way. We consider every possible state our newly introduced vertex v could have. Then for each of these possibilities we will compute, for every possible valid colouring of the rest of the vertices, the new entry in A_x. This results, for each possibility, in a slice of the new table A_x. When we have computed every slice we can join these together to form table A_x. Below we show how to compute each slice of A_x, for every possible state in S.

$$A_x(c \times \{2\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c(u) = 0_0 \\ 0 & \text{if } \kappa \leq 1 \\ A_y(c, \kappa - 2) & \text{otherwise} \end{cases}$$

$$A_x(c \times \{0_0\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c(u) = 2 \\ A_y(c, \kappa) & \text{otherwise} \end{cases}$$

$$A_x(c \times \{0_?\}, \kappa) = A_y(c, \kappa)$$

Since for state $0_?$ we are indifferent to the types of neighbours it has, we can just directly copy A_y into this slice. For states 0_0 and 2 we need to make sure that they are never neighbouring each other, since that is not allowed in a solution. So in any configuration where they are neighbours we set the number of solutions, $A_x(c, \kappa)$, to 0. For the state 2 we also have to make sure that we increase the cost by 2.

FORGET NODE:

Let x be a forget node in T with child node y that forgets the vertex v, and $c \in S^{X_y}$. To compute the result table for the forget node we essentially do the same as for the introduce node, but in reverse. We split our table A_y up in slices for each of the possible states and then sum these slices together to form the new table A_x. This can be done using the following equation, where we define $A_y(c \times \{0_0\}, -1)$ to be 0.

$$A_x(c, \kappa) = A_y(c \times \{2\}, \kappa) + A_y(c \times \{0_?\}, \kappa) - A_y(c \times \{0_0\}, \kappa) + A_y(c \times \{0_0\}, \kappa - 1)$$

We have to make sure that we only count assignments in which v is either secured or has a neighbour of state 2. This is because in a solution every unsecured vertex has a neighbour of state 2. If we make sure to only count these assignments, then in the root node (where we have forgotten every vertex) we will have exactly counted the solutions. We note that by simply summing every slice where $c(v) \in \{2, 0_?\}$, we get every possible assignment for v. Removing the slice where $c(v) = 0_0$ from this leaves us with only those assignments where $f(v) = 2$ or $\exists u \in N(v) : f(v) = 2$. Then we add the slice where $c(v) = 0_0$ with cost $\kappa - 1$, since we will count such a vertex v as a vertex for which $f(v) = 1$.

6.3 JOIN NODE:

$\times$	2	$0_?$	0_0
2	2		
$0_?$		$0_?$	
0_0			0_0

Table 1: The join table for Roman Domination.

Let x be a join node in T with child nodes l, r. Recall that the bags of each of these 3 nodes is the same, $X_r = X_l = X_x$. In a join node, like in the other types of nodes, we are trying to compute a memoisation table A_x. In order to do so we need to sum entries from A_l and A_r together to form a new table. For each entry in A_x we need to sum over those entries $A_l(c^l, \kappa)$ and $A_r(c^r, \kappa)$ that could have created this entry $A_x(c, \kappa)$. We note that to compute one such entry we need to add the entries from the memoisation tables from A_l and A_r that have colourings that agree with the colouring of the entry we are trying to compute.

In this case this is fairly straightforward since a vertex of state 2 in the resulting colouring has to have had state 2 in both the left and the right sub tree. The same is also true for all the other states. We can encode this information in a *join table*. A join table is a table in which we show for every combination of states from the left and right sub tree, what the resulting state in the resulting sub tree would be. Using the states set S this turns in to a diagonal join table as shown in 1.

A diagonal join table means that we can simply multiply entries from A_l and A_r together that have the same colouring and have costs that agree. This gives us the following equation.

$$A_x(c,\kappa) = \sum_{\kappa_l+\kappa_r=\kappa+2\#_2(c)} A_l(c,\kappa_l)\cdot A_r(c,\kappa_r)$$

We use the notation $\#_2$ here to indicate the number of vertices in the colouring of state 2. The sum is like this because in $\kappa_l + \kappa_r$ we count the vertices in the bag twice.

If we follow this algorithm, then, for the root node ρ, A_ρ will contain the number of solutions for every cost κ. Finally we note that every computation listed here can be done in polynomial time and we do each computation a polynomial amount of times for each entry in our memoisation table. Since our memoisation table is size $\mathcal{O}(3^t)$, the running time of this algorithm is $\mathcal{O}^*(3^t)$.

7 Treewidth Algorithm for Weak Roman Domination

In this section we will give an $\mathcal{O}^*(9^t)$ algorithm for Weak Roman Domination (Defensive Domination) for graphs of treewidth t. This algorithm can very easily be turned into an $\mathcal{O}^*(8^t)$ algorithm for Secure Domination (1-Capacitated Defensive Domination), by simply removing the value 2 from this algorithm. Our approach and notation is based on Chapter 11 from van Rooij [21].

7.1 Unique Defenders

In order to work towards a treewidth algorithm that solves the counting variant of Weak Roman Domination we start by introducing the notion of a unique defender. Intuitively a unique defender is the unique guard that defends a group of vertices.

Definition 7.1. (Unique Defender). Let f be a guard assignment, $G = (V, E)$ a graph and $v \in V$ an unsecured vertex. If $\sum_{w\in N(v)} f(w) = 1$, then there exists only one secured vertex $u \in N(v)$ and $f(u) = 1$. If this is the case we say that u is the *unique defender of* v, denoted by $\texttt{UD}(v) = u$. Otherwise we say that the unique defender of v does not exist, or $\texttt{UD}(v)$ is undefined. We also say that if $\texttt{UD}(v) = u$, then v is *uniquely defended by* u, u *uniquely defends* v, v is *uniquely defended* and u is a *unique defender*.

Definition 7.2. (Unique Defender Set). Let f be a guard assignment, $G = (V, E)$ a graph and $v \in V$ a vertex, for which $f(v) = 1$. We denote the unique defender set of v by $S_{\texttt{UD}}(v)$, where $S_{\texttt{UD}}(v)$ is the set of all vertices uniquely defended by v.

Note that if a vertex v has an empty unique defender set then v is also not a unique defender. Also note that, since any uniquely defended vertex has exactly one secured neighbour, these unique defender sets are disjoint.

Using these definitions we can characterise wrd-functions in some more local conditions. To do this we need the following 2 lemmas.

Lemma 7.3. *Let f be a guard assignment and $G = (V, E)$ a graph. If f is a weak roman domination function, then for every vertex $v \in V$, for which $f(v) = 1$, $S_{\texttt{UD}}(v)$ is a clique.*

Proof. Assume that there exists a vertex $v \in V$, for which $f(v) = 1$ and $S_{\texttt{UD}}(v)$ is not a clique. From Definitions 7.1 and 7.2 we know that every vertex in $S_{\texttt{UD}}(v)$ is in the neighbourhood of v. Also since $S_{\texttt{UD}}(v)$ is not a clique, there have to be at least 2 distinct vertices $u, w \in S_{\texttt{UD}}(v)$, for which $(u, w) \notin E$. Because u and w are both uniquely defended by v, both of these vertices are not safely defended (Definition 2.1), since $f_{v\to u}$ leaves w undefended and vice versa. In other words u is weakly defended by v and vice versa. This means that f is not a weak roman domination function. □

Lemma 7.4. *Let f be a guard assignment and $G = (V, E)$ a graph. f is a wrd-function if and only if every unique defender set is a clique and for every unsecured vertex $v \in V$ one of the following conditions is met.*

1. *v is uniquely defended, i.e., v is part of a unique defender set.*
2. *v is not uniquely defended and v has a secured neighbour that is not a unique defender.*
3. *v is not uniquely defended, and all secured neighbours of v are unique defenders, and for at least one secured neighbour u we have $S_{\texttt{UD}}(u) \subset N(v)$.*

Proof. The if part:

For every vertex $v \in V$ the following holds. v is either secured or unsecured. If v is secured it is also safely defended (Definition 2.1).

If v is unsecured then one of the 3 conditions is met. For all 3 cases we will show that there exists a secured neighbour u of v, such that $f_{u\to v}$ does not create an undefended vertex. First we note that the only vertices that can become undefended by $f_{u\to v}$ are those vertices in $N(u) \setminus v$.

- If Condition 1 is met then v is part of a unique defender set and v has a unique defender $u = \texttt{UD}(v)$, so u is the only secured neighbour of v. For all $w \in N(u) \setminus v$ the following holds. We know that if $w \notin S_{\texttt{UD}}(u)$ there exists a secured vertex $w' \in N(w) \setminus u$, since u is not the unique defender of w. This means that w is still defended by $f_{u\to v}$. We also know that $S_{\texttt{UD}}(u)$ is a clique, since every unique defender set is a clique. This means that if $w \in S_{\texttt{UD}}(u)$, then w is defended by $f_{u\to v}$, because w is a neighbour of v. Therefore v is safely defended.
- If Condition 2 is met then v has a secured neighbour u that is not a unique defender. This means that either $f(u) = 2$, which means that v is safely defended, or we know that for all $w \in N(u) \setminus v$ there exists a secured vertex $w' \in N(w) \setminus u$, since u is not a unique defender. Therefore all these vertices are still defended by $f_{u\to v}$, so v is safely defended.
- If Condition 3 is met, then v has a secured neighbour u that is a unique defender and $S_{\texttt{UD}}(u) \subset N(v)$. Following the same arguments as for Condition 1 we can see that all $w \in N(u) \setminus v$ are still defended by $f_{u\to v}$. We repeat these arguments here for completeness. We know that if $w \notin S_{\texttt{UD}}(u)$ there exists a secured vertex $w' \in N(w) \setminus u$, since u is not the unique defender of w. This means that w is still defended by $f_{u\to v}$. We also know that $S_{\texttt{UD}}(u)$ is completely in the neighbourhood of v. This means that if $w \in S_{\texttt{UD}}(u)$, then w is defended by $f_{u\to v}$, because w is a neighbour of v. This means that v is safely defended.

The only if part:

According to Lemma 7.3 every unique defender set is a clique, if f is a wrd-function. This proves the first part. For the second part we assume f is a wrd-function and there exists an unsecured vertex $v \in V$ for which none of the conditions hold. This means that v is not part of a unique defender set and one of the following is true.

1. v has no secured neighbours.
2. v has a set of secured neighbours U and all $u \in U$ are unique defenders, with $S_{\texttt{UD}}(u) \not\subset N(v)$

If the first case is true, v has no secured neighbours, so it is not defended, so f cannot be a wrd-function. If the second case is true, then for all $u \in U$ there exists a vertex $w \in S_{\texttt{UD}}(u)$ that is not in $N(v)$. This means that $f_{u\to v}$ will not defend w, so v is weakly defended by u and v is not safely defended. Therefore f cannot be a wrd-function. □

For any guard assignment $f : V \to \{2, 1, 0\}$, the vertices which are unique defenders (and uniquely defended) are uniquely defined and can be easily identified. This means that for every such f, there exists a unique $g : V \to \{2, 1, 1_{\texttt{UD}}, 0, 0_{\texttt{UD}}\}$ in which every vertex labelled $1_{\texttt{UD}}$ is a unique defender and every vertex labelled 1 is not (and every vertex labelled $0_{\texttt{UD}}$ is uniquely defended and every vertex labelled 0 is not). We will call such a function where the vertices are correctly labelled a *valid guard assignment* or if this assignment happens to be a solution a *valid wrd-function*.

Let X_x be a bag and $G_x = (V_x, E_x)$ its corresponding subgraph. Because a vertex can be a unique defender in one extension and not a unique defender in another (and the same for uniquely defended vertices), we use the function $g : V_x \to \{2, 1, 1_{\texttt{UD}}, 0, 0_{\texttt{UD}}\}$ for our partial guard assignments. We will call g a *valid partial guard assignment*, if and only if there exists an extension in which every vertex labelled $1_{\texttt{UD}}$ is a unique defender, every vertex labelled 1 is not, every vertex labelled $0_{\texttt{UD}}$ is uniquely defended and every vertex labelled 0 is not.

Lemma 7.5. *Let Let X_x be a bag, $G_x = (V_x, E_x)$ its corresponding subgraph and $g : V_x \to \{2, 1, 1_{UD}, 0, 0_{UD}\}$ a partial guard assignment. g is a valid partial guard assignment if and only if for every vertex $v \in V_x$ the following conditions are met.*

- *If $g(v) = 1_{UD}$, then $v \in X_x$ or v has a neighbour, u, for which $g(u) = 0_{UD}$.*
- *If $g(v) = 1$, then for every unsecured neighbour u of v, $g(u) = 0$.*
- *If $g(v) = 0_{UD}$, then v has no secured neighbours and $v \in X_x$, or v has exactly one secured neighbour u and $g(u) = 1_{UD}$.*
- *If $g(v) = 0$, then $v \in X_x$ or v has a neighbour u for which $g(u) = 2$ or v has more than one secured neighbour or v has no secured neighbours.*

Proof. Recall that g is a valid partial guard assignment, if and only if there exists an extension in which every vertex is correctly labelled. This means that we must prove the following. There exists an extension in which every vertex is correctly labelled, if and only if the conditions of this lemma are met.

The if part:

Let g be a partial guard assignment for which all of the conditions are met. We now consider the following extension H with guard assignment g'. For every vertex $v \in V_x$ we set $g'(v) = g(v)$. For every vertex $v \in X_x$ for which $g(v) = 1_{\texttt{UD}}$, we add a new vertex u (and an edge $\{u, v\}$) to H and we set $g'(u) = 0_{\texttt{UD}}$. For every vertex $v \in X_x$ that has no secured neighbours and for which $g(v) = 0_{\texttt{UD}}$, we add a new vertex u (and an edge $\{u, v\}$) to H and we set $g'(u) = 1_{\texttt{UD}}$. For every vertex $v \in X_x$ that has exactly one secured neighbour and for which $g(v) = 0$, we add a new vertex u (and an edge $\{u, v\}$) to H and we set $g'(u) = 2$.

In this extension H every vertex is correctly labelled by g', so there exists an extension on G_x in which every vertex is correctly labelled by g, thus g is a valid partial guard assignment.

The only if part:

Let g be a valid partial guard assignment. We will now show for each condition that if it is not met, we have a contradiction.

- If $g(v) = 1_{\texttt{UD}}$, $v \notin X_x$ and v has no neighbour u for which $g(u) = 0_{\texttt{UD}}$, then there exists no extension in which v has a neighbour u for which $g'(u) = 0_{\texttt{UD}}$. This means that there exists no extension in which v is correctly labelled.
- If $g(v) = 1$ and there exists a neighbour u of v for which $g(u) = 0_{\texttt{UD}}$, then either v is the unique defender of u and v is not correctly labelled or u is not uniquely defended and u is not correctly labelled. This means that there exists no extension in which every vertex is correctly labelled.
- If $g(v) = 0_{\texttt{UD}}$ and v has more than one secured neighbour, then v is not correctly labelled. If v has exactly one secured neighbour u and $g(u) \neq 1_{\texttt{UD}}$, then either v or u is not correctly labelled again. if v has no secured neighbours and $v \notin X_x$, then there exists no extension in which v has a neighbour w for which $g(w) = 1_{\texttt{UD}}$. This again means that there exists no extension in which every vertex is correctly labelled.
- If $g(v) = 0$, $v \notin X_x$, v has no neighbours u for which $g(u) = 2$ and v has exactly one secured neighbour, then by definition v is uniquely defended in every extension. This means that there exists no extension in which v is correctly labelled.

□

Let X_x be a bag, $G_x = (V_x, E_x)$ its corresponding subgraph and $g : V_x \to \{2, 1, 1_{\texttt{UD}}, 0, 0_{\texttt{UD}}\}$ a valid partial guard assignment. We can now define the unique defender set of a vertex v by $S_{\texttt{UD}}(v) = \{u \in X_x \cap N(v) \mid g(u) = 0_{\texttt{UD}}\}$. A valid partial guard assignment is a *valid partial wrd-function* if and only if there exists an extension for which this partial guard assignment can be turned into a valid wrd-function. Note that if we consider every possible valid partial wrd-function and their extensions, we will consider exactly the same wrd-functions as we would using f, since for every possible extension there exists exactly one valid guard assignment g for every guard assignment f.

Lemma 7.6. *Let X_x be a bag, $G_x = (V_x, E_x)$ its corresponding subgraph and $g : V_x \to \{2, 1, 1_{UD}, 0, 0_{UD}\}$ a valid partial guard assignment. g is a valid partial wrd-function if and only if every unique defender set is a clique and for every unsecured vertex $v \in V_x \setminus X_x$ one of the following conditions is met.*

1. *$g(v) = 0_{UD}$, i.e., v is uniquely defended.*
2. *$g(v) = 0$ and v has a neighbour u for which $g(u) \in \{2, 1\}$.*
3. *$g(v) = 0$, and for all secured neighbours u of v: $g(u) = 1_{UD}$, and for at least one secured neighbour u' we have $S_{UD}(u') \subset N(v)$.*

Proof. We consider the following extension H with guard assignment g'. For every vertex $v \in V_x$ for which $g(v) = 1_{\texttt{UD}}$ we do the following. If v is not a unique defender (i.e. $S_{\texttt{UD}}(v) = \emptyset$), then we add a new vertex w' in the extension for which we set $g'(w') = 0_{\texttt{UD}}$ and add an edge between v and w'. Next we add a new vertex w'', for which we set $g(w'') = 1_{\texttt{UD}}$ for each $u \in X_x$, for which $g(u) = 0_{\texttt{UD}}$ that does not have a secured neighbour and we add an edge between this w'' and u in our extension. Lastly we add a vertex w in our extension for which we set $g'(w) = 2$ and we add an edge between w and every vertex $u' \in X_x$, for which $g(u) = 0$.

We note that for H and g', the following holds. g' is a wrd-function if and only if g is a valid partial wrd-function. Also, every unique defender set in G_x is a clique and for every unsecured vertex in $V_x \setminus X_x$ one of the conditions is met if and only if every unique defender set in H is a clique and for every unsecured vertex in H one of the conditions from Lemma 7.4 is met. This means that we can apply Lemma 7.4, which proves this lemma. □

Using Lemma 7.6 we can start constructing a dynamic programming algorithm for weak roman domination. We will describe an algorithm that solves the counting variant of this problem as explained in Section 5.2. The problem we are trying to solve is therefore the following. Given a (nice) tree decomposition of a graph G of width t, find the number of wrd-functions of each cost κ, for $0 \leq \kappa \leq n$. Recall that the cost of a wrd-function f is defined as the total number of guards in this assignment, or $\sum_{v \in V} f(v)$. Also recall that according to Lemma 4.3 any variant of Defensive Domination has a solution of cost at most n, therefore it is sufficient to only consider solutions of costs smaller than or equal to n.

The algorithm we present will traverse the tree decomposition in a bottom up fashion, making sure that at each node x we store the number of valid partial solutions $g : V_x \to \{2, 1, 1_{\texttt{UD}}, 0, 0_{\texttt{UD}}\}$ of each cost κ (where the label $1_{\texttt{UD}}$ will have cost 1), for every colouring c. We will store this number in our memoisation table as $A_x(c, \kappa)$. To keep track of valid partial solutions we need to keep track of 2 things. First we have to keep track of the unique defender sets. Second we have to keep track of the unsecured vertices in our partial solution and what kind of secured neighbours they have. In particular we need to make sure that every unique defender set is always a clique and that for every vertex in $V_x \setminus X_x$ one of the conditions is met, so that Lemma 7.6 holds. In order to keep track of this we will need to define a suitable state set with which we can create a colouring of our vertices in the bag for a partial solution as explained in Section 5.2.

7.2 States

Let X_x be a bag, $G_x = (V_x, E_x)$ its corresponding subgraph and $g : V_x \to \{2, 1, 1_{\texttt{UD}}, 0, 0_{\texttt{UD}}\}$ a valid partial guard assignment. To make sure that every unique defender set is always a clique, we introduce the notion of active and inactive unique defender sets. Let v be a secured vertex. If $v \in X_x$ and $S_{\texttt{UD}}(v) \subset X_x$ we will call this set an *active unique defender set* of v, defined by $S_{\texttt{aUD}}(x, c, v) = \{u \in X_x \cap N(v) \mid c(u) = 0_{\texttt{UD}}\}$. When $v \notin X_x$, or when $S_{\texttt{UD}}(v) \not\subset X_x$ we will use the states $0^-_{\texttt{UD}}$ and $1^-_{\texttt{UD}}$ to represent this. We will call the part of the

unique defender set that is still in the bag an *inactive unique defender set* of v, with state $1^-_{\texttt{UD}}$, defined by $S_{\texttt{iUD}}(x, c, v) = \{u \in X_x \cap N(v) \mid c(u) = 0^-_{\texttt{UD}}\}$. We can then refer to the active and inactive unique defender of a vertex u, by $\texttt{aUD}(v)$ and $\texttt{iUD}(v)$ respectively. We will once again say that these functions are undefined if the specific unique defender does not exist. If we enforce the active unique defender set to always be a clique and we do not add any vertices to an inactive unique defender set over the course of our algorithm, we can ensure that every unique defender set is always a clique. However, to simplify the algorithm a little bit, we can delay the requirement of active unique defender sets being cliques. If we simply enforce these sets to be a clique when they turn inactive (i.e. when we forget a part of them) we know that they have to have been a clique all along. This means that we will knowingly store some configurations that are not partial solutions in our memoisation table, because we will eliminate them later in the algorithm.

Using the states we have introduced thus far $(2, 1, 1_{\texttt{UD}}, 1^-_{\texttt{UD}}, 0_{\texttt{UD}}, 0^-_{\texttt{UD}})$ we can ensure that every unique defender set will always be a clique and we can check whether Condition 1 or 3 of Lemma 7.6 is met. We can, however, not yet check if Condition 2 is met for a vertex v, for which $g(v) = 0$. To do this we need 4 more states. These states will encode the number of neighbours a vertex has, so that we can use it to see whether Condition 2 has been met. The first state 0_0 means that our vertex has no secured neighbours. Next, the state $\overline{0}$ means that our vertex has any number of secured neighbours, but all of them are unique defenders. Then we have the state 0_1, which means that our vertex has exactly one secured neighbour that is a unique defender and this neighbour is still in the bag. The state 0^-_1 means that our vertex has exactly one secured neighbour that is a unique defender and this neighbour is no longer in the bag. This distinction between 0_1 and 0^-_1 is only important in the join node of the treewidth algorithm. Lastly we have the state 0, which means that Condition 2 (or 3) of Lemma 7.6 is met for this vertex. One can imagine that using these 4 states we can count the number of (unique defender) neighbours a vertex has, to check whether Condition 2 is met. Using all of this we can finally define all of our states. We say that a vertex $v \in X_x$ has the following state if the description holds.

2	$g(v) = 2$.
1	$g(v) = 1$, and thus v is not a unique defender.
$1_{\texttt{UD}}$	$g(v) = 1_{\texttt{UD}}$ and $S_{\texttt{UD}}(v) \subset X_x$.
$1^-_{\texttt{UD}}$	$g(v) = 1_{\texttt{UD}}$ and $S_{\texttt{UD}}(v) \not\subset X_x$.
$0_{\texttt{UD}}$	$g(v) = 0_{\texttt{UD}}$ and $S_{\texttt{UD}}(u) \cup \{u\} \subseteq X_x$, where u is the unique defender of v.
$0^-_{\texttt{UD}}$	$g(v) = 0_{\texttt{UD}}$ and $S_{\texttt{UD}}(u) \cup \{u\} \not\subseteq X_x$, where u is the unique defender of v.
0	$g(v) = 0$ and for v either Condition 2 or Condition 3 of Lemma 7.6 is met.
$\overline{0}$	$g(v) = 0$ and for v none of the conditions of Lemma 7.6 are met and $\exists u \in N(v) : g(u) = 1_{\texttt{UD}}$.
0_1	$g(v) = 0$ and for v Condition 3 of Lemma 7.6 is not met and v does not have a unique defender, but v does have exactly one secured neighbour u, for which $g(u) = 1$ and $u \in X_x$.
0^-_1	$g(v) = 0$ and for v Condition 3 of Lemma 7.6 is not met and v does not have a unique defender, but v does have exactly one secured neighbour u, for which $g(u) = 1$ and $u \notin X_x$.
0_0	$g(v) = 0$ and all neighbours of v are unsecured, i.e. $\forall u \in N(v) : g(u) = 0$.

7.3 State Sets

The states defined in the previous section yield our original state set as follows.

$$S_0 = \{2, 1, 1_{\texttt{UD}}, 1^-_{\texttt{UD}}, 0, \overline{0}, 0_1, 0^-_1, 0_0, 0_{\texttt{UD}}, 0^-_{\texttt{UD}}\}$$

We can use this state set to define a colouring $c_0 : X_x \to S_0$. We will again slightly abuse notation when referring to such colourings. We will use a colouring c_i to refer to both the function that maps X_x to S_i, as well as the set of states that X_x is mapped to. This allows us to use the notation $c_i \times \{s\}$ to indicate

that we extend the colouring with a vertex v, for which $c_i(v) = s$. We note that this state set has 11 states which would, in a naive algorithm, result in an algorithm with a running time of at least $\mathcal{O}^*(11^t)$. It is, however, possible to determine whether a vertex has state 0_1, state 0_1^- or state $0_{\mathtt{UD}}$, by looking at the state of its neighbours in the bag. This is because a vertex of state 0_1 or state $0_{\mathtt{UD}}$, will need to have a neighbour of state 1 or state $1_{\mathtt{UD}}$ respectively and a vertex of state 0_1^- cannot have any secured neighbour. This means that our memoisation table containing values for every possible colouring using state set S_0 will always be sparse and contain $\mathcal{O}^*(9^t)$ elements. This also means that we can replace these 3 states by a single state, resulting in a state set of size 9. We will denote this new state by $\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle$, indicating that a vertex in this state represents a vertex that has one of the 3 states 0_1, 0_1^- or $0_{\mathtt{UD}}$. This results in our main state set, which we will use to store partial solutions.

$$S_{\mathtt{M}} = \{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, 0, \overline{0}, \langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle, 0_0, 0_{\mathtt{UD}}^-\}$$

In our algorithm we use colourings of the form $c_{\mathtt{M}} : X_x \to S_{\mathtt{M}}$ for entries in our memoisation table, $A_x(c_{\mathtt{M}}, \kappa)$. We can transform this table to a table using colourings of the form $c_0 : X_x \to S_0$ and back using the following transformations. We apply these transformations for a fixed κ and they can be repeated for each required κ. For these transformations we enforce some (arbitrary) order on the vertices in our bag X_x. These transformations then work in $|X_x|$ steps.

$\mathbf{S_0} \to \mathbf{S_M}$:

In the transformation from state set S_0 to $S_{\mathtt{M}}$ we say that, at step i, the first $|X_x| - i$ vertices in our bag are still being mapped to a state in S_0, while the last $i - 1$ vertices use the new state set $S_{\mathtt{M}}$. At this step we change, for the i-th vertex v, the state set it uses. We will therefore need to compute for every state in $S_{\mathtt{M}}$ the entry of A_x, where v has state $S_{\mathtt{M}}$. However, if v has a state in $\{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, 0, \overline{0}, 0_0, 0_{\mathtt{UD}}^-\}$, then we don't need to compute a new value, since these states are present in both sets. If v has state $\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle$ we apply the following transformation.

$$A_x(c_0 \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_{\mathtt{M}}, \kappa) = \sum_{s \in \{0_1, 0_1^-, 0_{\mathtt{UD}}\}} A_x(c_0 \times \{s\} \times c_{\mathtt{M}}, \kappa)$$

Since we consider for every vertex in the bag every possible state in $S_{\mathtt{M}}$ and we can compute the intermediate results in polynomial time, this transformation can be done in $\mathcal{O}^*(9^t)$ time.

$\mathbf{S_M} \to \mathbf{S_0}$:

The other way around we need to apply the following transformations.

$$A_x(c_{\mathtt{M}} \times \{0_{\mathtt{UD}}\} \times c_0) = A_x(c_{\mathtt{M}} \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_0, \kappa) \text{ if } \exists u \in N(v) : c_0 \times c_{\mathtt{M}}(u) = 1_{\mathtt{UD}}$$

$$A_x(c_{\mathtt{M}} \times \{0_1\} \times c_0) = A_x(c_{\mathtt{M}} \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_0, \kappa) \text{ if } \exists u \in N(v) : c_0 \times c_{\mathtt{M}}(u) = 1$$

$$A_x(c_{\mathtt{M}} \times \{0_1^-\} \times c_0) = A_x(c_{\mathtt{M}} \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_0, \kappa) \text{ otherwise}$$

In this case we compute entries using state set S_0. Since this state set has 11 states we cannot simply compute the entry for every possible colouring using this state set. We will do this by once again considering for every vertex in the bag every possible state in $S_{\mathtt{M}}$ and filling the corresponding value in the correct entry using state set S_0. This means that we can do this transformation in $\mathcal{O}^*(9^t)$ time as well.

In the first part of this algorithm we will use a couple of different state sets to allow us to achieve our running time. For three of the cases in an introduce node we need different state sets. These state sets S_2, S_1 and $S_{1_{\mathtt{UD}}}$ are for when we introduce a vertex of state 2, 1 and $1_{\mathtt{UD}}$ respectively. For state set S_2 we will replace state 0 with state $\langle 0\overline{0}0_1 0_1^- 0_0 \rangle$, while keeping the rest of the states the same. This gives us the following state set.

$$S_2 = \{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, \langle 0\overline{0}0_1 0_1^- 0_0 \rangle, \overline{0}, 0_1, 0_1^-, 0_0, 0_{\mathtt{UD}}, 0_{\mathtt{UD}}^-\}$$

In this new state set the state $\langle 0\overline{0}0_1 0_1^- 0_0 \rangle$ represents a vertex that is in one of the following states of the original state set $\{0, \overline{0}, 0_1, 0_1^-, 0_0\}$. We can therefore apply the following transformations to transform from S_0 to S_2 and back again.

$\mathbf{S_0 \to S_2}$

$$A_x(c_0 \times \{\langle 0\overline{0}0_1 0_1^- 0_0\rangle\} \times c_2, \kappa) = \sum_{s \in \{0,\overline{0},0_1,0_1^-,0_0\}} A_x(c_0 \times \{s\} \times c_2, \kappa)$$

$\mathbf{S_2 \to S_0}$

$$A_x(c_2 \times \{0\} \times c_0, \kappa) = A_x(c_2 \times \{\langle 0\overline{0}0_1 0_1^- 0_0\rangle\} \times c_0, \kappa) - \sum_{s \in \{\overline{0},0_1,0_1^-,0_0\}} A_x(c_2 \times \{s\} \times c_0, \kappa)$$

We note that state set S_2 has 11 states, just like state set S_0. However, since a table using S_0 is sparse and in S_2 we replace state 0, which did not play a role in the sparsity of S_0, we know that S_2 is also sparse. When performing the transformations above, we can make sure that we only do this for the valid entries in the table using S_0. Therefore we know that we can compute the sparse table using S_2 in $\mathcal{O}^*(9^t)$ time. The other way around we can use the same logic.

For state set S_1 we will replace the state 0 with the state $\langle 0\overline{0}0_1 0_1^-\rangle$. This yields the following state set.

$$S_1 = \{2, 1, 1_{\texttt{UD}}, 1_{\texttt{UD}}^-, \langle 0\overline{0}0_1 0_1^-\rangle, \overline{0}, 0_1, 0_1^-, 0_0, 0_{\texttt{UD}}, 0_{\texttt{UD}}^-\}$$

The new state $\langle 0\overline{0}0_1 0_1^-\rangle$ represents a vertex with a state in $\{0, \overline{0}, 0_1, 0_1^-\}$ in the original state set. For this state set we will use the following transformations to transform from and to the original state set.

$\mathbf{S_0 \to S_1}$

$$A_x(c_0 \times \{\langle 0\overline{0}0_1 0_1^-\rangle\} \times c_1, \kappa) = \sum_{s \in \{0,\overline{0},0_1,0_1^-\}} A_x(c_0 \times \{s\} \times c_1, \kappa)$$

$\mathbf{S_1 \to S_0}$

$$A_x(c_0 \times \{0\} \times c_1, \kappa) = A_x(c_0 \times \{\langle 0\overline{0}0_1 0_1^-\rangle\} \times c_1, \kappa) - \sum_{s \in \{\overline{0},0_1,0_1^-\}} A_x(c_0 \times \{s\} \times c_1, \kappa)$$

Using the same logic as for state set S_2, we can see that state set S_1 is also sparse and thus these transformations can be done in $\mathcal{O}^*(9^t)$ time.

For state set $S_{1_{\texttt{UD}}}$ we will replace the state $\overline{0}$ with the state $\langle \overline{0}0_0\rangle$ and state 0 with state $\langle 00_1\rangle$. This yields the following state set.

$$S_{1_{\texttt{UD}}} = \{2, 1, 1_{\texttt{UD}}, 1_{\texttt{UD}}^-, \langle 00_1\rangle, \langle \overline{0}0_0\rangle, 0_1, 0_1^-, 0_0, 0_{\texttt{UD}}, 0_{\texttt{UD}}^-\}$$

The new state $\langle \overline{0}0_0\rangle$ represents a vertex that is in state $\overline{0}$ or state 0_0 in the original state set. The new state $\langle 00_1\rangle$ represents a vertex that is in state 0 or state 0_1 in the original state set. For this state set we will use the following transformations to transform from and to the original state set.

$\mathbf{S_0 \to S_{1_{\texttt{UD}}}}$

$$A_x(c_0 \times \{\langle \overline{0}0_0\rangle\} \times c_{1_{\texttt{UD}}}, \kappa) = A_x(c_0 \times \{\overline{0}\} \times c_{1_{\texttt{UD}}}, \kappa) + A_x(c_0 \times \{0_0\} \times c_{1_{\texttt{UD}}}, \kappa)$$

$$A_x(c_0 \times \{\langle 00_1\rangle\} \times c_{1_{\texttt{UD}}}, \kappa) = A_x(c_0 \times \{0\} \times c_{1_{\texttt{UD}}}, \kappa) + A_x(c_0 \times \{0_1\} \times c_{1_{\texttt{UD}}}, \kappa)$$

$\mathbf{S_{1_{\texttt{UD}}} \to S_0}$

$$A_x(c_{1_{\texttt{UD}}} \times \{\overline{0}\} \times c_0, \kappa) = A_x(c_{1_{\texttt{UD}}} \times \{\langle \overline{0}0_0\rangle\} \times c_0, \kappa) - A_x(c_{1_{\texttt{UD}}} \times \{0_0\} \times c_0, \kappa)$$

$$A_x(c_{1_{\texttt{UD}}} \times \{0\} \times c_0, \kappa) = A_x(c_{1_{\texttt{UD}}} \times \{\langle 00_1\rangle\} \times c_0, \kappa) - A_x(c_{1_{\texttt{UD}}} \times \{0_1\} \times c_0, \kappa)$$

Again using the same logic as for state set S_2, we can see that state set $S_{1_{\texttt{UD}}}$ is also sparse and thus these transformations can be done in $\mathcal{O}^*(9^t)$ time.

These 3 state sets conclude the extra sets needed in the introduce nodes. In the forget nodes we need one more state set. In this state set we replace the state 0 with the state $\langle 0\overline{0}\rangle$. This leaves us with the following state set.

$$S_{\mathtt{F}} = \{2, 1, 1_{\mathtt{UD}}, 1^{-}_{\mathtt{UD}}, \langle 0\overline{0}\rangle, \overline{0}, 0_1, 0^{-}_1, 0_0, 0_{\mathtt{UD}}, 0^{-}_{\mathtt{UD}}\}$$

The new state $\langle 0\overline{0}\rangle$ represents a vertex that is in state 0 or state $\overline{0}$ in the original state set. We can use the following transformations to transform from and to the original state set.

$\mathbf{S_0} \to \mathbf{S_F}$

$$A_x(c_0 \times \{\langle 0\overline{0}\rangle\} \times c_{\mathtt{F}}, \kappa) = A_x(c_0 \times \{0\} \times c_{\mathtt{F}}, \kappa) + A_x(c_0 \times \{\overline{0}\} \times c_{\mathtt{F}}, \kappa)$$

$\mathbf{S_F} \to \mathbf{S_0}$

$$A_x(c_{\mathtt{F}} \times \{0\} \times c_0, \kappa) = A_x(c_{\mathtt{F}} \times \{\langle 0\overline{0}\rangle\} \times c_0, \kappa) - A_x(c_{\mathtt{F}} \times \{\overline{0}\} \times c_0, \kappa)$$

Once again we can use the same logic as for state set S_2, to see that state set $S_{\mathtt{F}}$ is also sparse and thus these transformations can be done in $\mathcal{O}^*(9^t)$ time.

7.4 Pathwidth Algorithm

Before we give the treewidth algorithm for Weak Roman Domination, we will start with a pathwidth algorithm. This is an algorithm that works on a path decomposition in stead of a tree decomposition. A nice path decomposition is essentially a nice tree decomposition without join nodes. Below we give a pathwidth algorithm that given a (nice) path decomposition of width at most t solves the counting problem for Weak Roman Domination in $\mathcal{O}^*(9^t)$ time (or the counting problem for Secure Domination in $\mathcal{O}^*(8^t)$ time).

We will show, for each type of node, that we can compute the table A_x, using state set $S_{\mathtt{M}}$. Here $A_x(c_{\mathtt{M}}, \kappa)$ represents the number of partial solutions to Weak Roman Domination of cost exactly κ satisfying the colouring $c_{\mathtt{M}}$.

LEAF NODE:

Let x be a leaf node in T. In the leaf nodes we only have empty bags and therefore the only possible colouring is $\emptyset$ with cost 0.

$$A_x(\emptyset, \kappa) = \begin{cases} 1 & \text{if } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

INTRODUCE NODE:

Let x be an introduce node in T with child node y that introduces the vertex v, and $c_{\mathtt{M}} \in {S_{\mathtt{M}}}^{X_y}$. In an introduce node we will first transform all our entries in the table A_y from state set $S_{\mathtt{M}}$ to S_0. Note that the resulting table will be a sparse table with $\mathcal{O}(9^t)$ entries. We will now start building up the result table A_x using state set S_0. We will compute every valid entry in A_x the same way as we did in the algorithm for Roman Domination. For the introduced vertex v we will consider every possible state it can have. For each of these possibilities we will compute for every possible valid colouring of the rest of the vertices the new entry in A_x. This results in a slice of the new table A_x. For every possible state v can have, using state set S_0, we compute the corresponding slice of A_x. After we have computed all the slices of A_x we can join all these slices together into our complete table A_x and then transform this table back to using state set $S_{\mathtt{M}}$. Below we show how to compute each slice of A_x, for every possible state in S_0.

When $c_0(v) = 2$, to guarantee our running time, we have to first transform A_y to state set S_2 before we compute A_x. After the transformation to S_2 we can compute A_x in the following way.

$$A_x(c_2 \times \{2\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_2(u) \in \{\overline{0}, 0_1, 0_1^-, 0_0, 0_{\mathtt{UD}}, 0_{\mathtt{UD}}^-\} \\ 0 & \text{if } \kappa \leq 1 \\ A_y(c_2, \kappa - 2) & \text{otherwise} \end{cases}$$

After this we transform this slice of A_x back to the original state set.

When the new vertex of state 2 is neighbouring a vertex with a state in $\{\overline{0}, 0_1, 0_1^-, 0_0, 0_{\mathtt{UD}}, 0_{\mathtt{UD}}^-\}$, then $A_x(c_0, \kappa)$ must be 0. Since all of these states cannot exist neighbouring a vertex of state 2. We can however place our vertex next to some of these states if we transform these states to state 0. We do not allow placing a vertex of state 2 next to a vertex of state $0_{\mathtt{UD}}$ or $0_{\mathtt{UD}}^-$, since we decided beforehand that these vertices had to be uniquely defended. The rest of the states in the list we will transform to state 0, when placing a vertex of state 2 next to it. This is done implicitly by the state transformations between S_0 and S_2. When we transform from S_0 to S_2, we gather all of these states together in the state $\langle 0\overline{0}0_1 0_1^- 0_0 \rangle$. During the computation of this slice of A_x we disallow the states $\{\overline{0}, 0_1, 0_1^-, 0_0\}$ to exist in the neighbourhood of v. Therefore, when we transform back to S_0, every colouring with a vertex of state $\langle 0\overline{0}0_1 0_1^- 0_0 \rangle$ in the neighbourhood of v will be counted as if this vertex has state 0.

When $c_0(v) = 1$ we will do this in the same way. First we transform A_y to state set S_1 and then use the following to compute A_x.

$$A_x(c_1 \times \{1\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_1(u) \in \{\overline{0}, 0_1^-, 0_0, 0_{\mathtt{UD}}, 0_{\mathtt{UD}}^-\} \\ 0 & \text{if } \kappa = 0 \\ A_y(c_1', \kappa - 1) & \text{otherwise} \end{cases}$$

Where c_1' is defined by:

- $\forall u \in X_y \setminus N(v) : c_1'(u) = c_1(u)$
- $\forall u \in X_y \cap N(v) :$
  $c_1(u) \in \{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, \langle 0\overline{0}0_1 0_1^- \rangle\}$ and $c_1'(u) = c_1(u)$,
  or $c_1(u) = 0_1$ and $c_1'(u) = 0_0$

Afterwards we transform this slice of A_x back to use S_0.

When we add a vertex of state 1 it is easy to see that any neighbouring vertex with a state in $\{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, 0\}$ will not be affected. However vertices of states 0_1, 0_1^- or $\overline{0}$ will turn into a vertex of state 0, because with this extra neighbour of state 1 Condition 2 of Lemma 7.6 is met (this is again done implicitly by the transformations). A vertex of state 0_0 will turn into state 0_1, since this new vertex will be its only neighbour with a state in $\{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-\}$. Neighbours of state $0_{\mathtt{UD}}$ or $0_{\mathtt{UD}}^-$ are again not allowed by definition.

When $c_0(v) = 1_{\mathtt{UD}}$ we will do this in the same way. First we transform A_y to state set $S_{1_{\mathtt{UD}}}$ and then use the following to compute A_x.

$$A_x(c_{1_{\mathtt{UD}}} \times \{1_{\mathtt{UD}}\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_{1_{\mathtt{UD}}}(u) \in \{0_{\mathtt{UD}}^-, 0_1, 0_1^-, 0_0\} \\ 0 & \text{if } \exists u \in N(v) : c_{1_{\mathtt{UD}}}(u) = 0_{\mathtt{UD}} \\ & \text{and } \exists w \in N(u) \setminus \{v\} : c_{1_{\mathtt{UD}}}(w) = 1_{\mathtt{UD}} \\ 0 & \text{if } \kappa = 0 \\ A_y(c_{1_{\mathtt{UD}}}, \kappa - 1) & \text{otherwise} \end{cases}$$

Afterwards we transform this slice of A_x back to use S_0.

We will always allow the introduction of a vertex of state $1_{\mathtt{UD}}$ even if this vertex does not yet have a neighbour of state $0_{\mathtt{UD}}$. We will enforce this in the forget node. This makes it so that we will consider every possible unique defender set. We will also allow any active unique defender set to exist in the bag even if

it is not a clique, since we will also enforce this in the forget node for simplicity. In the introduce node we only enforce every uniquely defended vertex to have at most one secured neighbour. Furthermore any neighbour with a state in $\{2, 1, 1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}, 0, \overline{0}, 0_{\mathtt{UD}}\}$ will not be affected and a neighbour of state 0_1 will turn into state 0 and a neighbour of state 0_0 will turn into state $\overline{0}$ (again this is done implicitly by the transformations).

For the rest of the cases we do not need any additional transformations so we will just list the rest of the computations below. We use the notation $\exists!$ here to indicate that there must exist a unique element for which the conditions hold.

$$A_x(c_0 \times \{1^-_{\mathtt{UD}}\}, \kappa) = 0$$

$$A_x(c_0 \times \{0\}, \kappa) = \begin{cases} A_y(c_0, \kappa) & \text{if } \exists u \in N(v) : c_0(u) = 2 \\ A_y(c_0, \kappa) & \text{if } \exists u, w \in N(v) : c_0(u) = 1 \wedge c_0(w) \in \{1, 1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(c_0 \times \{\overline{0}\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_0(u) \in \{1, 2\} \\ A_y(c_0, \kappa) & \text{if } \exists u \in N(v) : c_0(u) \in \{1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(c_0 \times \{0_1\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_0(u) \in \{2, 1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}\} \\ A_y(c_0, \kappa) & \text{if } \exists! u \in N(v) : c_0(u) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(c_0 \times \{0^-_1\}, \kappa) = 0$$

$$A_x(c_0 \times \{0_0\}, \kappa) = \begin{cases} 0 & \text{if } \exists u \in N(v) : c_0(u) \in \{2, 1, 1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}\} \\ A_y(c_0, \kappa) & \text{otherwise} \end{cases}$$

$$A_x(c_0 \times \{0_{\mathtt{UD}}\}, \kappa) = \begin{cases} A_y(c_0, \kappa) & \text{if } \exists! u \in N(v) : c_0(u) = 1_{\mathtt{UD}} \\ & \text{and } \forall w \in N(v) \setminus \{u\} : c_0(w) \notin \{2, 1, 1^-_{\mathtt{UD}}\} \\ A_y(c_0, \kappa) & \text{if } \forall u \in N(v) : c_0(u) \notin \{2, 1, 1_{\mathtt{UD}}, 1^-_{\mathtt{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(c_0 \times \{0^-_{\mathtt{UD}}\}, \kappa) = 0$$

For all of the rules above it is fairly easy to see that these are correct when looking at the definitions of the states. The only one that is more complex is when we introduce a vertex of state $0_{\mathtt{UD}}$. In this case we need to make sure that we only introduce it when it has at most one secured neighbour and this neighbour is state $1_{\mathtt{UD}}$.

FORGET NODE:

Let x be a forget node in T with child node y that forgets the vertex v, and $c_{\mathtt{M}} \in {S_{\mathtt{M}}}^{X_y}$. For simplicity we once again transform A_y to use state set S_0. After this we compute A_x as described below and transform this table back to use state set $S_{\mathtt{M}}$.

In a forget node we need to make sure that we only forget vertices in those states that are allowed in a valid solution. The states that we can simply forget without any extra work are $\{2, 1, 1^-_{\mathtt{UD}}, 0, 0^-_{\mathtt{UD}}\}$, since these vertices are either secured or one of the conditions from Lemma 7.6 is met. Furthermore we can also forget states $0_{\mathtt{UD}}$ and $1_{\mathtt{UD}}$. However since these states represent active unique defenders and active unique defender sets we have to check whether the set is a clique and mark the remainder as inactive when we forget these. We also have to check whether the unique defender set of a vertex with state $1_{\mathtt{UD}}$ is not empty, because this would mean that we count a non-unique defender as a unique defender, meaning that we would count solutions multiple times. Furthermore we have to check whether Condition 3 of Lemma 7.6 is met for some neighbouring vertices when we forget a vertex of state $0_{\mathtt{UD}}$ or $1_{\mathtt{UD}}$. Lastly we can also forget a vertex of state

$\overline{0}$, but only if for this vertex Condition 3 of Lemma 7.6 is met. If this is the case we have to choose one of its secured neighbours for which the unique defender set is in the neighbourhood of v and mark this set as inactive. When we mark this set as inactive we again have to check whether Condition 3 of Lemma 7.6 is met for some neighbouring vertices.

This all means that we can compute A_x using the following equation.

$$A_x(c_0, \kappa) = \sum_{s \in \{2,1,1^-_{\mathtt{UD}},1_{\mathtt{UD}},0,\overline{0},0^-_{\mathtt{UD}},0_{\mathtt{UD}}\}} a(c_0, \kappa, s)$$

Where a is defined differently for each s as we will define below. What we do here is essentially the same as for the introduce nodes but in reverse. We first split up our table in slices based on the state of v and then we compute A_x by adding these slices together. We will now list for every slice of A_y how to compute the corresponding part of the sum.

If $s \in \{2, 1, 1^-_{\mathtt{UD}}, 0, 0^-_{\mathtt{UD}}\}$ then

$a(c_0, \kappa, s) = A_y(c_0 \times \{s\}, \kappa)$

For all of the other cases we need to transform the slice of A_y to use state set $S_{\mathtt{F}}$ before we can do the computation to guarantee our running time. After we have done the computation we can transform the result back to use state set S_0 and add this to our result. Below we will list the computations in $S_{\mathtt{F}}$.

If $s = 1_{\mathtt{UD}}$ then let $U = S_{\mathtt{iUD}}(y, c_{\mathtt{F}}, v)$ in

$$a(c_{\mathtt{F}}, \kappa, 1_{\mathtt{UD}}) = \begin{cases} 0 & \text{if } U \text{ is not a clique} \\ 0 & \text{if } |U| = 0 \\ 0 & \text{if } \exists u \in X_y : c_{\mathtt{F}}(u) = \overline{0} \wedge \forall w' \in U \cup \{v\} : w' \in N(u) \\ A_y(c'_{\mathtt{F}} \times \{1_{\mathtt{UD}}\}, \kappa) & \text{otherwise} \end{cases}$$

Where $c'_{\mathtt{F}}$ is defined by:

- $\forall u \in X_y \setminus U : c'_{\mathtt{F}}(u) = c_{\mathtt{F}}(u)$
- $\forall u \in U : c'_{\mathtt{F}}(u) = 0_{\mathtt{UD}}$

If $s = 0_{\mathtt{UD}}$ then let $w = \mathtt{iUD}(v)$ and $U = S_{\mathtt{iUD}}(y, c_{\mathtt{F}}, w)$ in

$$a(c_{\mathtt{F}}, \kappa, 0_{\mathtt{UD}}) = \begin{cases} 0 & \text{if } w \text{ is undefined} \\ 0 & \text{if } U \text{ is not a clique} \\ 0 & \text{if } \exists u \in U : u \notin N(v) \\ 0 & \text{if } \exists u \in X_y : c_{\mathtt{F}}(u) = \overline{0} \wedge \forall w' \in U \cup \{v, w\} : w' \in N(u) \\ A_y(c'_{\mathtt{F}} \times \{0_{\mathtt{UD}}\}, \kappa) & \text{otherwise} \end{cases}$$

Where $c'_{\mathtt{F}}$ is defined by:

- $\forall u \in X_y \setminus (U \cup \{w\}) : c'_{\mathtt{F}}(u) = c_{\mathtt{F}}(u)$
- $\forall u \in U : c'_{\mathtt{F}}(u) = 0_{\mathtt{UD}}$
- $c'_{\mathtt{F}}(w) = 1_{\mathtt{UD}}$

If $s = \overline{0}$ then we have to consider every neighbouring inactive unique defender set, because each of these could have been turned inactive by forgetting v. Therefore we have to take a sum over all neighbouring inactive unique defenders. We can do this in the following way.

$$a(c_{\mathtt{F}}, \kappa, \overline{0}) = \sum_{w \in N(v) \cap X_y} a'(c_{\mathtt{F}}, \kappa, \overline{0}, w), \text{ where}$$

$$\text{let } U = S_{\mathtt{iUD}}(y, c_{\mathtt{F}}, w) \text{ in}$$

$$a'(c_{\mathtt{F}}, \kappa, \overline{0}, w) = \begin{cases} 0 & \text{if } c_{\mathtt{F}}(w) \neq 1^-_{\mathtt{UD}} \\ 0 & \text{if } U \text{ is not a clique} \\ 0 & \text{if } |U| = 0 \\ 0 & \text{if } \exists u \in U : u \notin N(v) \\ 0 & \text{if } \exists u \in X_y : c_{\mathtt{F}}(u) = \overline{0} \wedge \forall w' \in U \cup \{w\} : w' \in N(u) \\ A_y(c'_{\mathtt{F}} \times \{0_{\mathtt{UD}}\}, \kappa) & \text{otherwise} \end{cases}$$

Where $c'_{\mathtt{F}}$ is defined by:

- $\forall u \in X_y \setminus (U \cup \{w\}) : c'_{\mathtt{F}}(u) = c_{\mathtt{F}}(u)$
- $\forall u \in U : c'_{\mathtt{F}}(u) = 0_{\mathtt{UD}}$
- $c'_{\mathtt{F}}(w) = 1_{\mathtt{UD}}$

To compute A_x in the forget node we take a sum over $\mathcal{O}(t)$ elements for each of the $\mathcal{O}^*(9^t)$ elements in A_x. This means that we can compute this in $\mathcal{O}^*(9^t)$ time.

We note that one could argue that we should also transform vertices of state 0_1 to state 0^-_1 when we forget the neighbouring vertex of state 1. This is, however, not necessary since we will transform vertices in both these states to the state $\langle 0_1 0^-_1 0_{\mathtt{UD}} \rangle$, when we transform back to state set $S_{\mathtt{M}}$. This means that it does not matter if we do this extra transformation from state 0_1 to state 0^-_1, since the result we will store will remain the same (and correct).

Following this algorithm we again know that, for the root node ρ, A_ρ will contain the number of solutions for every cost κ. Finally we note that following this algorithm we consider every valid guard assignment $g : V \to \{2, 1, 1_{\mathtt{UD}}, 0, 0_{\mathtt{UD}}\}$ exactly once. This means that, because every valid solution g corresponds to exactly one solution $f : V \to \{2, 1, 0\}$, we find each of these exactly once. This means that the algorithm above solves the counting problem of Weak Roman Domination in $\mathcal{O}^*(9^t)$ time.

7.5 Treewidth Algorithm

To transform our pathwidth algorithm into a treewidth algorithm we only need to consider the join node. In this section we will explain what we need to do in this join node and how we can do this in $\mathcal{O}^*(9^t)$ time.

JOIN NODE:

7.5.1 Join Table

Let x be a join node in T with child nodes l, r. Recall that the bags of each of these 3 nodes is the same, $X_r = X_l = X_x$. Recall from our algorithm for Roman Domination, that in a join node, like in the other types of nodes, we are trying to compute a memoisation table A_x. We will repeat what we stated there to refresh what we are trying to do. In order to compute A_x we need to sum entries from A_l and A_r together to form a new table. For each entry in A_x we need to sum over those entries $A_l(c^l_{\mathtt{M}}, \kappa)$ and $A_r(c^r_{\mathtt{M}}, \kappa)$ that could have created this entry $A_x(c_{\mathtt{M}}, \kappa)$. We note that to compute one such entry we need to add the entries from the memoisation tables from A_l and A_r that have colourings that agree with the colouring of the entry we are trying to compute. This means that for every vertex in the bag the colourings $c^l_{\mathtt{M}}$ and $c^r_{\mathtt{M}}$ need to agree with $c_{\mathtt{M}}$. For example: if a vertex v has $c_{\mathtt{M}}(v) = 2$, then we know that both $c^l_{\mathtt{M}}(v) = 2$ and $c^r_{\mathtt{M}}(v) = 2$. This is because any vertex that has 2 guards will always have had 2 guards (i.e. there are no computations

$\times$	2	1	$1_{\mathtt{UD}}$	$1^-_{\mathtt{UD}}$	$0_{\mathtt{UD}}$	$0^-_{\mathtt{UD}}$	0_0	$\overline{0}$	0^-_1	0_1	0
2	2										
1		1									
$1_{\mathtt{UD}}$			$1_{\mathtt{UD}}$	$1^-_{\mathtt{UD}}$							
$1^-_{\mathtt{UD}}$			$1^-_{\mathtt{UD}}$								
$0_{\mathtt{UD}}$					$0_{\mathtt{UD}}$	$0^-_{\mathtt{UD}}$					
$0^-_{\mathtt{UD}}$					$0^-_{\mathtt{UD}}$						
0_0							0_0	$\overline{0}$	0^-_1		0
$\overline{0}$							$\overline{0}$	$\overline{0}$	0		0
0^-_1							0^-_1	0	0		0
0_1										0_1	0
0							0	0	0	0	0

Table 2: The join table using the original state set.

in either the introduce or forget nodes that change the number of guards on a vertex). This is also true for any vertex that has 1 or 0 guards, this vertex should have 1 or 0 guards, respectively, in all of the 3 nodes. This also means that $\sum_{v \in X_x} f(v) = \sum_{v \in X_l} f(v) = \sum_{v \in X_r} f(v)$ and thus that the cost of the vertices in the bag must be equal between all of the 3 nodes.

We note that if a vertex v has $c_{\mathtt{M}}(v) = 1$, then we also know that both $c^l_{\mathtt{M}}(v) = 1$ and $c^r_{\mathtt{M}}(v) = 1$, since a vertex that is not a unique defender may never have been a unique defender over the course of the algorithm. This is because we consider partial solutions of the form $g : V_x \to \{2, 1, 1_{\mathtt{UD}}, 0, 0_{\mathtt{UD}}\}$ and we therefore explicitly decide which vertices are unique defenders. If we consider a vertex, v that has $c_{\mathtt{M}}(v) = 1_{\mathtt{UD}}$ the same is still true (both $c^l_{\mathtt{M}}(v) = 1_{\mathtt{UD}}$ and $c^r_{\mathtt{M}}(v) = 1_{\mathtt{UD}}$). However if we consider a vertex v for which $c_{\mathtt{M}}(v) = 1^-_{\mathtt{UD}}$, then we know that it is not possible that both $c^l_{\mathtt{M}}(v) = 1^-_{\mathtt{UD}}$ and $c^r_{\mathtt{M}}(v) = 1^-_{\mathtt{UD}}$. This is because this would mean that, in both the left and the right sub tree, at least one vertex of the corresponding unique defender set has been forgotten. This means that these 2 forgotten vertices cannot have an edge between them, which means that this unique defender set cannot be a clique. Therefore we know that, for v, $c^l_{\mathtt{M}}(v) = 1_{\mathtt{UD}}$ and $c^r_{\mathtt{M}}(v) = 1^-_{\mathtt{UD}}$ or $c^l_{\mathtt{M}}(v) = 1^-_{\mathtt{UD}}$ and $c^r_{\mathtt{M}}(v) = 1_{\mathtt{UD}}$. We can create a table for every possible combination of states, from the left and right sub tree, for a vertex and then fill this with the state this would result in in the resulting colouring. This will again give us a *join table* and the table for this algorithm is shown in Table 2 using state set S_0.

We can see, in this table, that for example a vertex v for which $c_0(v) = 0^-_1$ in the left bag and $c_0(v) = \overline{0}$ in the right bag, must have $c_0(v) = 0$ in the resulting bag. This is because this vertex has a neighbour of state $1_{\mathtt{UD}}$ in the right sub tree and a vertex of state 1 in the left sub tree. An empty square in this table means that these 2 states cannot be joined. For all of the entries in this table it is fairly easily checked that they are indeed correct. We do want to pay a little extra attention to the unique defender sets.

As we discussed before we can join 2 active unique defender sets (states $0_{\mathtt{UD}}$ and $1_{\mathtt{UD}}$), but not 2 inactive unique defender sets (states $0^-_{\mathtt{UD}}$ and $1^-_{\mathtt{UD}}$). Since we always turn the entire unique defender set from active to inactive at once (and thus a vertex of state $1_{\mathtt{UD}}$ cannot have a neighbour of state $0^-_{\mathtt{UD}}$) we know the following. When we follow our join table we will always join 2 active unique defender sets (into an active unique defender set) or one active unique defender set and one inactive unique defender set (into an inactive unique defender set). When in one of the 2 colourings a unique defender set is inactive we know that this set is a clique in all 3 bags, because the bags are all the same. When a unique defender set is active in both bags it does not have to be a clique yet, because we will enforce this when we turn this set inactive later in the algorithm.

We could directly use this join table to compute the resulting memoisation table in our join node. However, this would mean that we need to consider $\mathcal{O}^*(27^t)$ possible combinations between the left and the right bag (27 is the number of non empty squares in the join table). This would mean that the running time of our algorithm will be similar. This is something we want to avoid. We note that in an optimal scenario our join table would be diagonal, since then the number of non empty squares is equal to the number of states. In our case we also want to be able to do the join using 2 states less than the original state set, because the original state set has 11 states. This means that we would ideally want to do the join using a state set like $S_{\mathtt{M}}$, while also fixing the problem of the non diagonal join table.

$\times$	$1_{\mathtt{UD}}$	$1^-_{\mathtt{UD}}$
$1_{\mathtt{UD}}$	$1_{\mathtt{UD}}$	$1^-_{\mathtt{UD}}$
$1^-_{\mathtt{UD}}$	$1^-_{\mathtt{UD}}$	

$\times$	0_0	$\overline{0}$	0^-_1	0_1	0
0_0	0_0	$\overline{0}$	0^-_1		0
$\overline{0}$	$\overline{0}$	$\overline{0}$	0		0
0^-_1	0^-_1	0	0		0
0_1				0_1	0
0	0	0	0	0	0

$\times$	$0_{\mathtt{UD}}$	$0^-_{\mathtt{UD}}$
$0_{\mathtt{UD}}$	$0_{\mathtt{UD}}$	$0^-_{\mathtt{UD}}$
$0^-_{\mathtt{UD}}$	$0^-_{\mathtt{UD}}$	

Table 3: The 3 parts of the join table that need to be optimised.

7.5.2 Improved Method

We will solve the problems presented in the previous section in the following way. When we take a closer look at the join table, we can see that there are 3 separate parts that are not yet diagonal. These 3 parts are placed side by side in Table 3. We will diagonalize all 3 of these parts using indexation techniques. The indexation techniques presented here are a combination of the techniques used for Perfect Matching and σ-ρ-Domination in Chapter 11 of [19].

The idea behind these techniques is best explained using the left most table from Table 3. What we do to perform this part of the join is transform our memoisation table to use a state set with the state $1^-_{\mathtt{UD}}$ replaced by the state $\langle 1_{\mathtt{UD}} 1^-_{\mathtt{UD}} \rangle$. This new state represents a vertex that is either in state $1_{\mathtt{UD}}$ or in state $1^-_{\mathtt{UD}}$. When we join 2 vertices of this state together the resulting vertex has been created from one of 4 combinations of states from the left and right sub tree, namely $1_{\mathtt{UD}}1_{\mathtt{UD}}$, $1_{\mathtt{UD}}1^-_{\mathtt{UD}}$, $1^-_{\mathtt{UD}}1_{\mathtt{UD}}$ or $1^-_{\mathtt{UD}}1^-_{\mathtt{UD}}$. If we would then transform back to our main state set we would subtract all the solutions where this vertex has state $1_{\mathtt{UD}}$, which can only be created by the combination $1_{\mathtt{UD}}1_{\mathtt{UD}}$ in the left and right sub tree. This would almost give us a result that agrees with the left most table from Table 3, except that we now also count the combination $1^-_{\mathtt{UD}}1^-_{\mathtt{UD}}$. This is the problem we solve with the indexation technique. Before we perform our first transformation we count, for every colouring in our memoisation table, the number of vertices of state $1^-_{\mathtt{UD}}$ and index this colouring with this number. This allows us to perform the join and then extract those entries in the resulting table for which the number of vertices of state $1^-_{\mathtt{UD}}$ is exactly equal to the number of vertices of state $1^-_{\mathtt{UD}}$ in the left sub tree plus the number of vertices of state $1^-_{\mathtt{UD}}$ in the right sub tree. This eliminates the possibility of counting the combination $1^-_{\mathtt{UD}}1^-_{\mathtt{UD}}$, since this would mean that the sum of the counts between the left and right sub tree is larger than the count in the resulting tree. For the right most table in 3, the technique is exactly the same. For the middle table the idea is still the same, but the execution is now a little more complex. We show that our technique gives a result that agrees with the entire join table in Section 7.5.3. We do that in a similar fashion as we did here for states $1_{\mathtt{UD}}$ and $1^-_{\mathtt{UD}}$.

We will now present the state sets, computations and indexation techniques needed to compute the join node in $\mathcal{O}^*(9^t)$ time. First we introduce the state set in which we need to compute the join. This state set is the following.

$$S_4 = \{2, 1, 1_{\mathtt{UD}}, \langle 1_{\mathtt{UD}}1^-_{\mathtt{UD}}\rangle, \langle 0_{\mathtt{UD}}0^-_{\mathtt{UD}}\rangle, 0_0, \langle 0_0\overline{0}\rangle, \langle 0_0 0_1 0^-_1 0_{\mathtt{UD}}\rangle, \langle 0_0\overline{0}0_1 0^-_1 0\rangle\}$$

This state set has 9 states, where the new states represent vertices that have one of a number of states from the original state set, similar to the state sets used in the introduce and forget nodes. For now we will assume that we can transform from state set $S_{\mathtt{M}}$ to S_4 in $\mathcal{O}^*(9^t)$ time using transformations similar to how we handled the other state sets in the introduce and forget nodes. The actual transformations are listed in Section 7.5.4.

Next we will present our indexes. Let $y \in \{l, r\}$ and let A_y be the memoisation tables using state set $S_{\mathtt{M}}$. We will expand our tables A_y and index them by 3 indexes, namely i^1, i^0 and j. Indexes i^1 and i^0 represent the number of vertices in X_y, with states $1^-_{\mathtt{UD}}$ and $0^-_{\mathtt{UD}}$ respectively. The index j will represent the number of vertices of state 0^-_1 that have been transformed to state $\langle 0_0 0_1 0^-_1 0_{\mathtt{UD}}\rangle$ later on. The indexes i^1 and i^0 serve to diagonalize the left most and right most table from Table 3 respectively. The index j serves to diagonalize the middle table. The indexed tables are computed as follows.

$$A'_y(c_{\mathtt{M}}, \kappa, i^1, i^0, j) = \begin{cases} A_y(c_{\mathtt{M}}, \kappa) & \text{if } \#_{1^-_{\mathtt{UD}}}(c_{\mathtt{M}}) = i^1 \text{ and } \#_{0^-_{\mathtt{UD}}}(c_{\mathtt{M}}) = i^0 \text{ and } j = 0 \\ 0 & \text{otherwise} \end{cases}$$

Here we use the # notation to indicate the number of vertices of the specific state in the bag. We initialise

the index j to be 0 for every entry in our tables. We will use this, in our transformations, to count the number of vertices in state 0_1^- that are transformed to $\langle 0_0 0_1 0_1^- 0_{\mathtt{UD}} \rangle$. The resulting tables A'_y are $\mathcal{O}(k^3)$ times as large as the original tables, but these indexed tables allow us to compute any entry in the joined table A'_x in $\mathcal{O}(nk^3)$ time, which means that we can compute the entire joined table A'_x in $\mathcal{O}^*(9^t)$ time.

In order to compute this joined table we first need to transform our tables A'_y to use state set S_4 (during which we index our table correctly with j). We will then compute the joined table A'_x from A'_l and A'_r using the following formula.

$$A'_x(c_4, \kappa, i^1, i^0, j) = \sum_{\kappa_l + \kappa_r = \kappa + \#_{1^*}(c_4) + 2\#_2(c_4)} \left(\sum_{i^0 = i_l^0 + i_r^0} \sum_{i^1 = i_l^1 + i_r^1} \sum_{j = j_l + j_r} A'_l(c_4, \kappa_l, i_l^1, i_l^0, j_l) \cdot A'_r(c_4, \kappa_r, i_r^1, i_r^0, j_r) \right)$$

Here we again use the # notation introduced earlier, with a similar meaning. However here $\#_{1^*}(c)$ represents the number of vertices with a state in $\{1, \langle 1_{\mathtt{UD}} 1_{\mathtt{UD}}^- \rangle\}$. We use the first sum here to make sure that we count those solutions that have a cost that agrees with our result, keeping in mind that with $\kappa_l + \kappa_r$ we count the vertices in the bag twice. During these computations we make sure that the indexes in the resulting table A'_x are always a sum of the indexes from our tables A'_y. We do this because we know that the number of vertices of state $1_{\mathtt{UD}}^-$ in the resulting colouring has to be equal to the number of vertices of state $1_{\mathtt{UD}}^-$ in the left colouring plus the number of vertices of state $1_{\mathtt{UD}}^-$ in the right colouring. This is because a vertex of state $1_{\mathtt{UD}}^-$ can only be created by a vertex of the same state in one of the 2 colourings, but a vertex of state $1_{\mathtt{UD}}$ in the other. The same argument also holds for the other indexes.

This means that the table A'_x now contains all the correct entries we need to construct our final table A_x. It is however not using the correct state set for us to extract the final table. We will therefore start to transform our state set back to our main state set. Before we do this, however, we need one last intermediate state set, to allow us to correctly do the extraction. In this state set we transform all of the states from state set S_4 back to states from state set $S_{\mathtt{M}}$ except for the state $\langle 0_0 \overline{0} 0_1 0_1^- 0 \rangle$, which we keep the same. The resulting state set is the following.

$$S_5 = \{2, 1, 1_{\mathtt{UD}}, 1_{\mathtt{UD}}^-, 0_{\mathtt{UD}}^-, 0_0, \overline{0}, \langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle, \langle 0_0 \overline{0} 0_1 0_1^- 0 \rangle\}$$

This state set again has 9 states, where the new states represent vertices that have one of a number of states from the original state set. For now we will again assume that we can transform from state set S_4 to S_5 and from S_5 to $S_{\mathtt{M}}$ in $\mathcal{O}^*(9^t)$ time using transformations similar to how we handled the other state sets in the introduce and forget nodes. These transformations are again listed in Section 7.5.4.

After transforming A'_x to use state set S_5 we can extract the final table A_x. To do this we simply take all those values that are indexed correctly according to the number of vertices of each state in the corresponding colouring. This gives us the following equation.

$$A_x(c_5, \kappa) = A'_x(c_5, \kappa, \#_{1_{\mathtt{UD}}^-}(c_5), \#_{0_{\mathtt{UD}}^-}(c_5), \#_{0_1^-}(c_5))$$

We again use the # notation here. The functions $\#_{1_{\mathtt{UD}}^-}(c_5)$ and $\#_{0_{\mathtt{UD}}^-}(c_5)$ are the same as defined earlier. The function $\#_{0_1^-}(c_5)$ is defined a little different, since we are counting the number of occurrences of a state from a different state set as the one our colouring is using. We can still define this since we can differentiate between these states based on the neighbours in the following way.

$$\#_{0_1^-}(c_5) = \sum_{v \in X_x} \begin{cases} 1 & \text{if } c_5(v) = \langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle \text{ and } \forall u \in N(v) \cap X_x : c_5(u) \notin \{1, 1_{\mathtt{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

After we have extracted the table A_x all that is left to do is transform this table back to use state set $S_{\mathtt{M}}$, which is something that we can do in $\mathcal{O}^*(9^t)$ time as shown in Section 7.5.4. In Section 7.5.3 we will show that the result we get with this approach is the same as the result that simply using the join table would have gotten us. If we add this approach for the join node to our pathwidth algorithm we once again know that the root node will contain for every cost the number of solutions.

$0_0 0_0$	$0_1^- 0_0$	$0_1 0_0$	$0_{\mathtt{UD}} 0_0$
$0_0 0_1^-$	$0_1^- 0_1^-$	$0_1 0_1^-$	$0_{\mathtt{UD}} 0_1^-$
$0_0 0_1$	$0_1^- 0_1$	$0_1 0_1$	$0_{\mathtt{UD}} 0_1$
$0_0 0_{\mathtt{UD}}$	$0_1^- 0_{\mathtt{UD}}$	$0_1 0_{\mathtt{UD}}$	$0_{\mathtt{UD}} 0_{\mathtt{UD}}$

$0_0 0_0$	$0_1^- 0_0$		
$0_0 0_1^-$	$0_1^- 0_1^-$		
		$0_1 0_1$	
			$0_{\mathtt{UD}} 0_{\mathtt{UD}}$

	$0_1^- 0_0$		
$0_0 0_1^-$			
		$0_1 0_1$	
			$0_{\mathtt{UD}} 0_{\mathtt{UD}}$

Table 4: All possible combinations of states from the left and right sub tree in state set S_0 that can result in state $\langle 0_0 0_1 0_1^- 0_{\mathtt{UD}} \rangle$ in the join. On the left we have every possible combination, in the middle we have all possible combinations that remain after removing the first 10 and on the right we have all combinations that remain after transforming to $\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle$.

$0_0 0_0$	$\overline{0} 0_0$
$0_0 \overline{0}$	$\overline{0}\,\overline{0}$

	$\overline{0} 0_0$
$0_0 \overline{0}$	$\overline{0}\,\overline{0}$

Table 5: All possible combinations of states from the left and right sub tree in state set S_0 that can result in state $\langle 0_0 \overline{0} \rangle$ in the join. On the left we have every possible combination and on the right we have all combinations that remain after transforming to $\overline{0}$.

7.5.3 Correctness

In the join node we first introduce a number of states that represent several states from the original state set. Next we just directly join the colourings that are the same in state set S_4. By doing so we actually join all possible combinations of the represented states from the original state set. We will now show that doing this, combined with the indexing techniques and transformations back to $S_{\mathtt{M}}$, will agree with the result of Table 2. First, we look at the state set in which we compute the join, S_4. This set has 4 states that have a direct correspondence to a state in S_0. This means that these states will just be joined one on one. These states are 2, 1, $1_{\mathtt{UD}}$ and 0_0. Looking at Table 2 we see that this is the result we want for these states.

Next we have the state $\langle 0_0 0_1 0_1^- 0_{\mathtt{UD}} \rangle$. This is one of the most complex parts of the join. In the left and right sub tree a vertex in this state represents a vertex in one of the states 0_0, 0_1, 0_1^- and $0_{\mathtt{UD}}$ from state set S_0. This means that a vertex that has this state after the join can be created from one of the 16 combinations between states from the left and right sub tree shown in Table 4. From these 16 combinations we can directly eliminate 10, because we join the states 1 and $1_{\mathtt{UD}}$ one on one. In other words we require X_l and X_r to have exactly the same vertices with these states. This means that for example the combination $0_1, 0_0$ cannot happen. This combination tells us that we have a vertex that has exactly one neighbours of state 1 in the left sub tree, but this same vertex has no such neighbours in the right sub tree. This is not possible. Using this same logic we can eliminate these 10 combinations $0_1 0_0$, $0_{\mathtt{UD}} 0_0$, $0_1 0_1^-$, $0_{\mathtt{UD}} 0_1^-$, $0_0 0_1$, $0_1^- 0_1$, $0_{\mathtt{UD}} 0_1$, $0_0 0_{\mathtt{UD}}$, $0_1^- 0_{\mathtt{UD}}$ and $0_1 0_{\mathtt{UD}}$. This leaves us with the second table in Table 4. From these combinations we subtract the combination $0_0 0_0$ when we transform to state set S_5 and finally we only extract the values where sum of the number of vertices of state 0_1^- between the left and the right sub tree that have been transformed to $\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle$ is the same as the number of vertices of state 0_1^- in the resulting colouring. This means that a vertex of state 0_1^- cannot have been created by state 0_1^- in both the left and the right sub tree, because then this sum would be higher than this number. This also means that a vertex of state $\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle$ cannot have been created by state 0_1^- in both the left and the right sub tree. This leaves us with only the combinations in the right most table in Table 4. These are also exactly the combinations we want, looking at Table 2.

Next we look at the state $\langle 0_0 \overline{0} \rangle$. For this state we show every possible combination in the left table of Table 5. From this we simply subtract the combination $0_0 0_0$ when we transform to state set S_5, directly resulting in the right table.

Then we have the states $\langle 0_{\mathtt{UD}} 0_{\mathtt{UD}}^- \rangle$ and $\langle 1_{\mathtt{UD}} 1_{\mathtt{UD}}^- \rangle$. Both of these are handled the same way. We subtract the combinations $0_{\mathtt{UD}} 0_{\mathtt{UD}}$ and $1_{\mathtt{UD}} 1_{\mathtt{UD}}$ when we transform to state set S_4. Afterwards we extract only those values where the numbers of vertices of states $0_{\mathtt{UD}}^-$ and $1_{\mathtt{UD}}^-$ in the resulting colouring are equal to the sum of these numbers in the colourings of the left and right sub tree. This eliminates the combinations $0_{\mathtt{UD}}^- 0_{\mathtt{UD}}^-$ and $1_{\mathtt{UD}}^- 1_{\mathtt{UD}}^-$. The resulting possible combinations again agree with Table 2.

Lastly we have the state $\langle 0_0 \overline{0} 0_1 0_1^- 0 \rangle$. In Table 7 on the left we see all possible combinations that can create a vertex of state $\langle 0_0 \overline{0} 0_1 0_1^- 0 \rangle$. As shown before there are some combinations in this table that can never happen, namely $0_1 0_0$, $0_1 \overline{0}$, $0_1 0_1^-$, $0_0 0_1$, $\overline{0} 0_1$, $0_1^- 0_1$. So we can remove these from the table. When we

$0_{\text{UD}}0_{\text{UD}}$	$0_{\text{UD}}0^-_{\text{UD}}$
$0^-_{\text{UD}}0_{\text{UD}}$	$0^-_{\text{UD}}0^-_{\text{UD}}$

$0_{\text{UD}}0^-_{\text{UD}}$
$0^-_{\text{UD}}0_{\text{UD}}$

$1_{\text{UD}}1_{\text{UD}}$	$1_{\text{UD}}1^-_{\text{UD}}$
$1^-_{\text{UD}}1_{\text{UD}}$	$1^-_{\text{UD}}1^-_{\text{UD}}$

$1_{\text{UD}}1^-_{\text{UD}}$
$1^-_{\text{UD}}1_{\text{UD}}$

Table 6: All possible combinations of states from the left and right sub tree in state set S_0 that can result in state $\langle 0_{\text{UD}}0^-_{\text{UD}}\rangle$ and state $\langle 1_{\text{UD}}1^-_{\text{UD}}\rangle$ in the join respectively. Each of the 2 pairs of tables shows on the left every possible combination and on the right all combinations that remain after transforming.

$0_0 0_0$	$\overline{0}_0$	$0^-_1 0_0$	$0_1 0_0$	00_0
$0_0\overline{0}$	$\overline{0}\overline{0}$	$0^-_1\overline{0}$	$0_1\overline{0}$	$0\overline{0}$
$0_0 0^-_1$	$\overline{0}0^-_1$	$0^-_1 0^-_1$	$0_1 0^-_1$	00^-_1
$0_0 0_1$	$\overline{0}0_1$	$0^-_1 0_1$	$0_1 0_1$	00_1
$0_0 0$	$\overline{0}0$	$0^-_1 0$	$0_1 0$	00

				00_0
		$0^-_1\overline{0}$		$0\overline{0}$
	$\overline{0}0^-_1$	$0^-_1 0^-_1$		00^-_1
				00_1
$0_0 0$	$\overline{0}0$	$0^-_1 0$	$0_1 0$	00

Table 7: All possible combinations of states from the left and right sub tree in state set S_0 that can result in state $\langle 0_0\overline{0}0_1 0^-_1 0\rangle$ in the join. On the left we have every possible combination and on the right we have all combinations that remain after transforming to state 0.

transform this state to state 0 we subtract the states 0_0, $\overline{0}$, 0_1 and 0^-_1. Removing all the combinations that are represented by these states we are left with the right table in Table 7. This table again agrees with Table 2.

7.5.4 Transformations

In this section we will list all of the transformations we need to perform when computing the join node. To transform from and to state set S_4 we need an intermediate state set. We will call this state set $S_{\hat{4}}$, defined as follows.

$$S_{\hat{4}} = \{2, 1, 1_{\text{UD}}, 1^-_{\text{UD}}, \langle 0_{\text{UD}}0^-_{\text{UD}}\rangle, 0_0, \langle 0_0\overline{0}\rangle, \langle 0_0 0_1 0^-_1 0_{\text{UD}}\rangle, \langle 0_0\overline{0}0_1 0^-_1 0\rangle\}$$

This state set is almost the same as state set S_4, with the exception that the state $\langle 1_{\text{UD}}1^-_{\text{UD}}\rangle$ is replaced by the original state 1^-_{UD}. We need this state set because during our transformations we need to check whether a vertex has a neighbour of state 1_{UD} and if we also have a state that represents a vertex of either state 1_{UD} or state 1^-_{UD}, this is not well defined.

The first transformation we will show now is the transformation from state set S_{M} to state set S_4. We can do this transformation by first transforming to the intermediate state set $S_{\hat{4}}$ and then on to state set S_4. We will do so using the following list of computations. Where v is the vertex currently being transformed. To concisely define this transformation we will use the following function.

$$a_1(s) = \begin{cases} A'_y(c_{\text{M}} \times \{s\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) & \text{if } s \in S_{\text{M}} \\ A'_y(c_{\text{M}} \times \{\langle 0_1 0^-_1 0_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) & \text{if } s = 0_{\text{UD}} \text{ and } \exists u \in N(v) \cap X_y : c_{\text{M}} \times c_{\hat{4}}(u) = 1_{\text{UD}} \\ A'_y(c_{\text{M}} \times \{\langle 0_1 0^-_1 0_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) & \text{if } s = 0_1 \text{ and } \exists u \in N(v) \cap X_y : c_{\text{M}} \times c_{\hat{4}}(u) = 1 \\ A'_y(c_{\text{M}} \times \{\langle 0_1 0^-_1 0_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) & \text{if } s = 0^-_1 \text{ and } \forall u \in N(v) \cap X_y : c_{\text{M}} \times c_{\hat{4}}(u) \notin \{1, 1_{\text{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{S_M \to S_{\hat{4}}}$:
$A'_y(c_{\text{M}} \times \{\langle 0_{\text{UD}}0^-_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) = a_1(0^-_{\text{UD}}) + a_1(0_{\text{UD}})$
$A'_y(c_{\text{M}} \times \{\langle 0_0\overline{0}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) = a_1(0_0) + a_1(\overline{0})$
$A'_y(c_{\text{M}} \times \{\langle 0_0 0_1 0^-_1 0_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) = a_1(0_0) + a_1(0_1) + a_1(0_{\text{UD}})$
$A'_y(c_{\text{M}} \times \{\langle 0_0 0_1 0^-_1 0_{\text{UD}}\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j+1) = a_1(0^-_1)$
$A'_y(c_{\text{M}} \times \{\langle 0_0\overline{0}0_1 0^-_1 0\rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) = \sum_{s \in \{0_0, \overline{0}, 0, 0_1, 0^-_1\}} a_1(s)$

$\mathbf{S_{\hat{4}} \to S_4}$:
$A'_y(c_{\hat{4}} \times \{\langle 1_{\text{UD}}1^-_{\text{UD}}\rangle\} \times c_4, \kappa, i^1, i^0, j) = A'_y(c_{\hat{4}} \times \{1^-_{\text{UD}}\} \times c_4, \kappa, i^1, i^0, j) + A'_y(c_{\hat{4}} \times \{1_{\text{UD}}\} \times c_4, \kappa, i^1, i^0, j)$

This transformation is a lot more extensive than the transformations for the introduce and forget nodes, but

essentially it works in the same way. Using this transformation we can compute every entry in our resulting table in polynomial time, meaning that we can do the entire transformation in $\mathcal{O}^*(9^t)$ time.

Next we will show the transformation from state set S_4 to state set S_5. For this transformation we again first transform to the intermediate state set $S_{\hat{4}}$ and then on to state set S_5. We can do this transformation using the computations listed below. For this transformation we again define a function, a_2, for conciseness.

$$a_2(s) = \begin{cases} A'_x(c_{\hat{4}} \times \{s\} \times c_5, \kappa, i^1, i^0, j) & \text{if } s \in S_4 \\ A'_x(c_{\hat{4}} \times \{\langle 0_0 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_5, \kappa, i^1, i^0, j) & \text{if } s = 0_{\mathtt{UD}} \text{ and } \exists u \in N(v) \cap X_x : c_{\hat{4}} \times c_5(u) = 1_{\mathtt{UD}} \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{S_4 \to S_{\hat{4}}}$:
$A'_x(c_4 \times \{1^-_{\mathtt{UD}}\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) = A'_x(c_4 \times \{\langle 1_{\mathtt{UD}} 1^-_{\mathtt{UD}} \rangle\} \times c_{\hat{4}}, \kappa, i^1, i^0, j) - A'_x(c_4 \times \{1_{\mathtt{UD}}\} \times c_{\hat{4}}, \kappa, i^1, i^0, j)$

$\mathbf{S_{\hat{4}} \to S_5}$:
$A'_x(c_{\hat{4}} \times \{0^-_{\mathtt{UD}}\} \times c_5, \kappa, i^1, i^0, j) = a_2(\langle 0_{\mathtt{UD}} 0^-_{\mathtt{UD}} \rangle) - a_2(0_{\mathtt{UD}})$
$A'_x(c_{\hat{4}} \times \{\overline{0}\} \times c_5, \kappa, i^1, i^0, j) = a_2(\langle 0_0 \overline{0} \rangle) - a_2(0_0)$
$A'_x(c_{\hat{4}} \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_5, \kappa, i^1, i^0, j) = a_2(\langle 0_0 0_1 0_1^- 0_{\mathtt{UD}} \rangle) - a_2(0_0)$

Again, using this transformation we can compute every entry in our resulting table in polynomial time, meaning that we can do the entire transformation in $\mathcal{O}^*(9^t)$ time.

This only leaves the last transformation from state set S_5 to state set $S_{\mathtt{M}}$. For this transformation we do not need an intermediate state set, but we will define a function, a_3, for conciseness.

$$a_3(s) = \begin{cases} A_x(c_5 \times \{s\} \times c_{\mathtt{M}}, \kappa) & \text{if } s \in S_5 \\ A_x(c_5 \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_{\mathtt{M}}, \kappa) & \text{if } s = 0_1 \text{ and } \exists u \in N(v) \cap X_x : c_5 \times c_{\mathtt{M}}(u) = 1 \\ A_x(c_5 \times \{\langle 0_1 0_1^- 0_{\mathtt{UD}} \rangle\} \times c_{\mathtt{M}}, \kappa) & \text{if } s = 0_1^- \text{ and } \forall u \in N(v) \cap X_x : c_5 \times c_{\mathtt{M}}(u) \notin \{1, 1_{\mathtt{UD}}\} \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{S_5 \to S_{\mathtt{M}}}$:
$A_x(c_5 \times \{0\} \times c_{\mathtt{M}}, \kappa) = a_3(\langle 0_0 \overline{0} 0_1 0_1^- 0 \rangle) - \sum_{s \in \{0_0, \overline{0}, 0_1, 0_1^-\}} a_3(s)$

Again, using this transformation we can compute every entry in our resulting table in polynomial time, meaning that we can do the entire transformation in $\mathcal{O}^*(9^t)$ time.

This concludes our treewidth algorithm for Weak Roman Domination. We once again note that this algorithm would also work for Secure Domination in $\mathcal{O}^*(8^t)$ time, by simply removing the state 2.

References

[1] R. Burdett and M. Haythorpe. An improved binary programming formulation for the secure domination problem. *ArXiv*, 2019, arXiv:1911.02198.

[2] A. P. Burger, E. J. Cockayne, W. R. Gründlingh, C. M. Mynhardt, J. H. van Vuuren, and W. Winterbach. Infinite order domination in graphs. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 50:179–194, 2004.

[3] A. P. Burger, A. P. De Villiers, and J. H. van Vuuren. A binary programming approach towards achieving effective graph protection. In *ORSSA*, pages 19–30, 2013.

[4] A. P. Burger, A. P. De Villiers, and J. H. Van Vuuren. Two algorithms for secure graph domination. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 85:321–339, 2013.

[5] M. Chapelle, M. Cochefert, J. F. Couturier, D. Kratsch, R. Letourneur, M. Liedloff, and A. Perez. Exact algorithms for weak Roman domination. *Discrete Applied Mathematics*, 248:79–92, 2018.

[6] E. J. Cockayne, P. A. Dreyer, S. M. Hedetniemi, and S. T. Hedetniemi. Roman domination in graphs. *Discrete Mathematics*, 278(1-3):11–22, 2004.

[7] E. J. Cockayne, P. J. P. Grobler, W. R. Grundlingh, J. Munganga, and J. H. van Vuuren. Protection of a Graph. *Utilitas Mathematica*, 67:19–32, 2005.

[8] H. Fernau. Roman domination: a parameterized perspective. *International Journal of Computer Mathematics*, 85(1):25–38, 2008.

[9] W. Goddard, S. M. Hedetniemi, and S. T. Hedetniemi. Eternal Security in Graphs. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 52:1–12, 2005.

[10] M. A. Henning and S. T. Hedetniemi. Defending the Roman Empire - A new strategy. *Discrete Mathematics*, 266(1-3):239–251, 2003.

[11] Y. Iwata. A Faster Algorithm for Dominating Set Analyzed by the Potential Method. In D. Marx and P. Rossmanith, editors, *Parameterized and Exact Computation*, pages 41–54, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] T. Kloks. *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media, 1994.

[13] W. F. Klostermeyer and C. M. Mynhardt. Protecting a graph with mobile guards. *Applicable Analysis and Discrete Mathematics*, 10(1):1–29, 2016.

[14] J. Nederlof and J. M. M. van Rooij. Inclusion/Exclusion Branching for Partial Dominating Set and Set Splitting. In V. Raman and S. Saurabh, editors, *Parameterized and Exact Computation*, pages 204–215, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[15] S.-L. Peng and Y.-H. Tsai. Roman domination on graphs of bounded treewidth. In *Proceedings of the 24th Workshop on Combinatorial Mathematics and Computation Theory*, pages 128–131, 2007.

[16] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[17] Z. Shi and K. M. Koh. Counting the Number of Minimum Roman Dominating Functions of a Graph. *ArXiv*, 2014, 1403.1019.

[18] A. Tripathi. Six Ways to Count the Number of Integer Compositions. *Crux mathematicorum*, 39(2):84–88, 2013.

[19] J. M. M. van. Rooij. *Exact Exponential-Time Algorithms for Domination Problems in Graphs*. PhD thesis, University Utrecht, 2011.

[20] J. M. M. van Rooij. Fast algorithms for join operations on tree decompositions. In F. V. Fomin, S. Kratsch, and E. J. van Leeuwen, editors, *Treewidth, Kernels, and Algorithms: Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, pages 262–297. Springer International Publishing, Cham, 2020.

[21] J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution. In A. Fiat and P. Sanders, editors, *Algorithms - ESA 2009*, pages 566–577, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.